

**Method for Allocating Memory Space for Limited  
Packet Head and/or Tail Growth**

*Inventor(s)*

*Enric Musoll Stephen Melvin Mario Nemirovsky*

5

**Field of the Invention**

10

The present invention is in the field of digital processing and pertains to apparatus and methods for processing packets in routers for packet networks, and more particularly to apparatus and methods for allocating enough memory to packet storage for enabling packet growth during processing in dynamic multistreaming processors.

15

**Cross-Reference to Related Documents**

20

The present invention is a continuation in part (CIP) to a U.S. patent application S/N 09/737,375 entitled "***Queuing System for Processors in Packet Routing Operations***" and filed on 12/14/00, which is incorporated herein in its entirety by reference. In addition, S/N 09/737,375 claims priority benefit under 35 U.S.C. 119 (e) of Provisional Patent Application serial number 60/181,364 filed on 2/8/2000, which is also incorporated in its entirety by reference. The present application is also a CIP of patent application serial number 09/602,279 filed on 06/23/2000 and is incorporated herein in its entirety by reference. The inventors have also participated in the document disclosure program and claim priority to document disclosure number 492429.

25

30

### **Background of the Invention**

5 The well-known Internet network is a notoriously well-known publicly-accessible communication network at the time of filing the present patent application, and arguably the most robust information and communication source ever made available. The Internet is used as a prime example in the present application of a data-packet-network which will benefit from the apparatus and methods taught in the present patent application, but is just one such network, following a particular standardized protocol. As is also very well known, the Internet (and related networks) are always a work in progress. That is, many researchers and developers are competing at all times to provide new and better apparatus and methods, including software, for enhancing the operation of such networks.

10 In general the most sought-after improvements in data packet networks are those that provide higher speed in routing (more packets per unit time) and better reliability and fidelity in messaging. What is generally needed are router apparatus and methods increasing the rates at which packets may be processed in a router.

15 As is well-known in the art, packet routers are computerized machines wherein data packets are received at any one or more of typically multiple ports, processed in some fashion, and sent out at the same or other ports of the router to continue on to downstream destinations. As an example of such computerized operations, keeping in mind that the Internet is a vast interconnected network of individual routers, individual routers have to keep track of which external routers to which they are connected by communication ports, and of which of alternate routes through the network are the best routes for incoming packets. Individual routers must also

20

25

accomplish flow accounting, with a flow generally meaning a stream of packets with a common source and end destination. A general desire is that individual flows follow a common path. The skilled artisan will be aware of many such requirements for computerized processing.

Typically a router in the Internet network will have one or more Central Processing Units (CPUs) as dedicated microprocessors for accomplishing the many computing tasks required. In the current art at the time of the present application, these are single-streaming processors; that is, each processor is capable of processing a single stream of instructions. In some cases developers are applying multiprocessor technology to such routing operations. The present inventors have been involved for some time in development of dynamic multistreaming (DMS) processors, which processors are capable of simultaneously processing multiple instruction streams. One preferred application for such processors is in the processing of packets in packet networks like the Internet.

In a data-packet processor, a configurable queuing system for packet accounting during processing is known to the inventor and disclosure for same is referenced herein as S/N 09/737,375 in the Cross-Reference section of this specification. The queuing and accounting system has a plurality of queues arranged in one or more clusters, an identification mechanism for creating a packet identifier for arriving packets, insertion logic for inserting packet identifiers into queues and for determining into which queue to insert a packet identifier, and selection logic for selecting packet identifiers from queues to initiate processing of identified packets, downloading of completed packets, or for re-queuing of the selected packet identifiers.

A portion of the memory in the above-described system is called packet memory. The packet memory is the memory where data packets reside before they can be downloaded by a packet management unit (PMU)

to an output network interface (ONI) during packet processing and routing.

A portion of the packet memory described above is called the local packet memory (LPM), and it is directly managed by hardware in the PMU instead of by software.

5           Whenever a data packet has been processed and is ready to be downloaded from LPM, the processing core or streaming processor unit (SPU) sends a command *PKTDONE* to the PMU. This command contains, among other information, a packet identifier (typically a number) of the packet that is ready to be downloaded. The PMU will then proceed with the  
10 download of this packet if it resides in LPM. If not, software operating through a system interface unit (SIU) will download the packet upon request.

          Data packet processing occurs using context registers. In a dynamic multistreaming processor running eight streams, there are at least eight context registers for processing data packets. The context registers are located physically within the streaming processor core having associated therewith functional units required for processing. In this example, each context can have a state of either PMU-owned or SPU-owned. When information is being preloaded into a context the context is PMU-owned.  
15  
20 When a stream is running within a context, the context is SPU-owned.

          During packet processing wherein the SPU is executing instructions to process a data packet, it is possible that the data packet in process will be caused to grow (become larger) in terms of header data and/or tail data. Packet growth during SPU processing may cause the process to overwrite  
25 consecutively stored data packets information in memory. The problem presented by this possible growth phenomena is that in order to avoid a possible overwrite and data corruption issue, a packet that has outgrown its memory space must be moved during processing to a new memory space

that is large enough (larger than before) to hold it in its expanded form. Repeated instances of this requirement over time tend to add more latency in packet processing. Therefore, what is clearly needed is a method for allocating enough memory space for a new packet to account for a possible growth in packet size during processing. A method such as this would sharply reduce packet move operations, and avoid overwrites in memory.

### **Summary of the Invention**

In a preferred embodiment of the present invention a hardware/software system for storing and downloading a data packet is provided, comprising a first configuration register for storing an offset value representing an amount of memory preceding the first line of the data packet to be stored, a second configuration register for storing an offset value representing an amount of memory following the end of the last line of the data packet to be stored, and a mechanism for allocating a portion of memory for storage of data packet. The system is characterized in that the offsets indicated by the first and second configuration registers are added to size of the data packet to be stored, resulting in a new size for memory allocation.

In a preferred embodiment the system is implemented within a data packet router connected to a data packet network, and the network may be the Internet network. In some embodiments the hardware accesses the first and second configuration registers, computes the required size of memory allocation, and allocates the memory for packet storage. In some other embodiments the software accesses the first and second registers, computes the required size of memory allocation, and allocates the memory for packet

storage. In still other embodiments the hardware accesses the first and second registers and passes the values to the software to complete the computation and allocation of the memory to store the data packet.

5 In some embodiments of the invention there are further one or more additional sets of registers for storing memory values applicable and specific to data packet types, each set associated with the type of data packet other than a type associated with the first and second registers. In some other embodiments the first and second registers are configurable during a boot operation of the system. There may also be one or more additional sets of registers that are configurable during a boot operation of the system. In still 10 other embodiments change values for the first and second registers may be entered into the registers during operation of the system. In the cases where there are additional configuration registers, the change values for the one or more additional sets of registers may be entered into the registers during operation of the system.

15 In some embodiments any changes in packet size during processing are tracked, and the first configuration register is consulted before download of the packet, to determine the actual starting point of the processed packet in memory.

20 In another aspect of the system method for storing and downloading a data packet is provided, comprising the steps of (a) providing an offset value in a first configuration register representing an amount of memory preceding the first line of the data packet to be stored; (b) providing an offset value in a second configuration register representing an amount of memory following the last line of the data packet to be stored (c) upon 25 arrival of a data packet to be stored, accessing the first and second configuration registers for the offset values; (d) summing the acquired

values with the original size of the data packet to be stored; and (e) allocating memory for the packet according to the sum in step (d).

In some preferred embodiments the system comprises a data packet router connected to a data packet network, and the network may be the well-known Internet network.

In some embodiments, in steps (a) and (b) the registers are accessible by hardware responsible for storing the data packet. In other embodiments in steps (a) and (b) the values are configured into each register during a boot operation of a host system.

In some embodiments in step (c) access is performed by hardware, and in some other in step (c) access is performed by software. There may further be an additional step between steps (c) and (d) for passing the values to software. In some embodiments steps (c) through (e) are performed entirely by hardware, or step (c) is performed by hardware and steps (d) and (e) are performed by software.

In some embodiments in steps (a) and (b) the registers are configured during boot operation, and the memory is local packet memory that is hardware controlled. Further, in other embodiments, in steps (a) and (b) there are a plurality of register sets, each set dedicated to a different type of data packet.

In yet other embodiments there are further steps for tracking growth and shrink of a data packet during processing, and for consulting the first configuration register after processing and before download to determine the actual starting point of the processed packet in memory.

The embodiments of the invention described in enabling detail below, for the first time provide an ability in a packet processor to allocate memory in a manner allowing for packet growth and shrinkage.

**Brief Description of the Drawings**

Fig. 1 is a simplified block diagram showing relationship of functional areas of a DMS processor in a preferred embodiment of the present invention.

Fig. 2 is a block diagram of the DMS processor of Fig. 1 showing additional detail.

Fig. 3 is a block diagram illustrating uploading of data into the LPM or EPM in an embodiment of the invention.

Fig. 4a is a diagram illustrating determination and allocation for data uploading in an embodiment of the invention.

Fig. 4b is a diagram showing the state that needs to be maintained for each of the four 64KB blocks.

Figs. 5a and 5b illustrate an example of how atomic pages are allocated in an embodiment of the present invention.

Figs. 6a and 6b illustrate how memory space is efficiently utilized in an embodiment of the invention.

Fig. 7 is a top-level schematic of the blocks of the XCaliber PMU unit involved in the downloading of a packet.

Fig. 8 is a diagram illustrating the phenomenon of packet growth and shrink.

Fig. 9 is a block diagram showing high-level communication between the QS and other blocks in the PMU and SPU in an embodiment of the present invention.

Fig. 10 is a table illustrating six different modes in an embodiment of the invention into which the QS can be configured.



Fig. 11 is a diagram illustrating generic architecture of the QS of Figs. 2 and 7 in an embodiment of the present invention.

Fig. 12 is a table indicating coding of the outbound DeviceId field in an embodiment of the invention.

5 Fig. 13 is a table illustrating priority mapping for RTU transfers in an embodiment of the invention.

Fig. 14 is a table showing allowed combinations of Active, Completed, and Probed bits for a valid packet in an embodiment of the invention.

10 Fig. 15 is a Pattern Matching Table in an embodiment of the present invention.

Fig. 16 illustrates the format of a mask in an embodiment of the invention.

Fig. 17 shows an example of a pre-load operation using the mask in Fig. 16.

15 Fig. 18 illustrates shows the PMU Configuration Space in an embodiment of the present invention.

Figs. 19a, 19b and 19c are a table of Configuration register Mapping.

Fig. 20 is an illustration of a PreloadMaskNumber configuration register.

20 Fig. 21 illustrates a PatternMatchingTable in a preferred embodiment of the present invention.

Fig. 22 illustrates a VirtualPageEnable configuration register in an embodiment of the invention.

25 Fig. 23 illustrates a ContextSpecificPatternMatchingMask configuration register in an embodiment of the invention.

Fig. 24 illustrates the MaxActivePackets configuration register in an embodiment of the present invention.

Fig. 25 illustrates the TimeCounter configuration register in an embodiment of the present invention.

Fig. 26 illustrates the StatusRegister configuration register in an embodiment of the invention.

5 Fig. 27 is a schematic of a Command Unit and command queues in an embodiment of the present invention.

Fig. 28 is a table showing the format of command inserted in command queues in an embodiment of the present invention.

10 Fig. 29 is a table showing the format for responses that different blocks generate back to the CU in an embodiment of the invention.

Fig. 30 shows a performance counter interface between the PMU and the SIU in an embodiment of the invention.

Fig. 31 shows a possible implementation of internal interfaces among the different units in the PMU in an embodiment of the present invention.

15 Fig. 32 is a diagram of a BypassHooks configuration register in an embodiment of the invention.

Fig. 33 is a diagram of an InternalStateWrite configuration register in an embodiment of the invention.

20 Figs. 34-39 comprise a table listing events related to performance counters in an embodiment of the invention.

Fig. 40 is a table illustrating the different bypass hooks implemented in the PMU in an embodiment of the invention.

Fig. 41 is a table relating architecture and hardware blocks in an embodiment of the present invention.

25 Figs. 42-45 comprise a table showing SPU-PMU Interface in an embodiment of the invention.

Figs. 46-49 comprise a table showing SIU-PMU Interface in an embodiment of the invention.

Fig. 50 is a block diagram illustrating a packet overwrite problem due to head and tail growth of the packet during processing.

Fig. 51. is a block diagram illustrating insulative packet queuing according to an embodiment of the present invention.

5

### **Description of the Preferred Embodiments**

10

15

20

25

In the provisional patent application S/N 60/181,364 referenced above there is disclosure as to the architecture of a DMS processor, termed by the inventors the XCaliber processor, which is dedicated to packet processing in packet networks. Two extensive diagrams are provided in the referenced disclosure, one, labeled NIO Block Diagram, shows the overall architecture of the XCaliber processor, with input and output ports to and from a packet-handling ASIC, and the other illustrates numerous aspects of the Generic Queue shown in the NIO diagram. The NIO system in the priority document equates to the Packet Management Unit (PMU) in the present specification. It is to the several aspects of the generic queue that the present application is directed.

Fig. 1 is a simplified block diagram of an XCaliber DMS processor 101 with a higher-level subdivision of functional units than that shown in the NIO diagram of the priority document. In Fig. 1 XCaliber DMS processor 101 is shown as organized into three functional areas. An outside System Interface Unit (SIU) area 107 provides communication with outside devices, that is, external to the XCaliber processor, typically for receiving and sending packets. Inside, processor 101 is divided into two broad functional units, a Packet Management Unit (PMU) 103, equating to the NIO system in

30

the priority document mentioned above, and a Stream Processor Unit (SPU) 107. The functions of the PMU include accounting for and managing all packets received and processed. The SPU is responsible for all computational tasks.

5           The PMU is a part of the XCaliber processor that offloads the SPU from performing costly packet header accesses and packet sorting and management tasks, which would otherwise seriously degrade performance of the overall processor.

10           Packet management is achieved by (a) Managing on-chip memory allocated for packet storage, (b) Uploading, in the background, packet header information from incoming packets into different contexts (context registers, described further below) of the XCaliber processor, (c) Maintaining, in a flexible queuing system, packet identifiers of the packets currently in process in the XCaliber.

15           The described packet management and accounting tasks performed by the PMU are performed in parallel with processing of packets by the SPU core. To implement this functionality, the PMU has a set of hardware structures to buffer packets incoming from the network, provide them to the SPU core and, if needed, send them out to the network when the processing is completed. The PMU features a high degree of programmability of several of its functions, such as configuration of its internal packet memory storage and a queuing system, which is a focus of the present patent application.

20           Fig. 2 is a block diagram of the XCaliber processor of Fig. 1 showing additional detail. SIU 107 and SPU 105 are shown in Fig. 2 as single blocks with the same element numbers used in Fig. 1. The PMU is shown in considerably expanded detail, however, with communication lines shown between elements.

In Fig. 2 there is shown a Network/Switching Fabric Interface 203 which is in some cases an Application Specific Integrated Circuit (ASIC) dedicated for interfacing directly to a network, such as the Internet for example, or to switching fabric in a packet router, for example, receiving and transmitting packets, and transacting the packets with the XCaliber processor. In this particular instance there are two in ports and two out ports communicating with processor 201. Network in and out interface circuitry 205 and 215 handle packet traffic onto and off the processor, and these two interfaces are properly a part of SIU 107, although they are shown separately in Fig. 2 for convenience.

Also at the network interface within the PMU there are, in processor 201, input and output buffers 207 and 217 which serve to buffer the flow of packets into and out of processor 201.

Referring again to Fig. 1, there is shown a Packet Management Unit (PMU) 103, which has been described as a unit that offloads the requirement for packet management and accounting from the Stream Processing Unit. This is in particular the unit that has been expanded in Fig. 2, and consists substantially of Input Buffer (IB) 207, Output Buffer (OB) 217, Paging Memory Management Unit (PMMU) 209, Local Packet Memory (LPM) 219, Command Unit (CU) 213, Queueing System (QS) 211, Configuration Registers 221, and Register Transfer Unit (RTU) 227. The communication paths between elements of the PMU are indicated by arrows in Fig. 2, and further description of the elements of the PMU is provided below, including especially QS 211, which is a particular focus of the present patent application.

## **Overview of PMU**

Again, Fig. 2 shows the elements of the PMU, which are identified briefly above. Packets arrive to the PMU in the present example through a 16-byte network input interface. In this embodiment packet data arrives to the PMU at a rate of 20 Gbps (max). At an operating speed of 300MHz XCaliber core frequency, an average of 8 bytes of packet data are received every XCaliber core cycle. The incoming data from the network input interface is buffered in InBuffer (IB) block 207. Network interface 205 within XCaliber has the capability of appending to the packet itself the size of the packet being sent, in the event that the external device has not been able to append the size to the packet before sending the packet. Up to 2 devices can send packet data to XCaliber at (10Gbps per device), and two in ports are shown from an attached ASIC. It is to be understood that the existence and use of the particular ASIC is exemplary, and packets could be received from other devices. Further, there may be in some embodiments more or fewer than the two in ports indicated.

Packet Memory Manager Unit (PMMU) 209 decides whether each incoming packet has to be stored into on-chip Local Packet Memory (LPM) 219, or, in the case that, for example, no space exists in the LPM to store it, may decide to either send the packet out to an External Packet Memory (EPM) not shown through the SIU block, or may decide to drop the packet. In case the packet is to be stored in the LPM, the PMMU decides where to store the packet and generates all the addresses needed to do so. The addresses generated correspond in a preferred embodiment to 16-byte lines in the LPM, and the packet is consecutively stored in this memory.

In the (most likely) case that the PMMU does not drop the incoming packet, a packet identifier is created, which includes a pointer (named packetPage) to a fixed-size page in packet memory where the packet has started to be stored. The identifier is created and enqueued into Queuing System (QS) block 211. The QS assigns a number from 0 to 255 (named packetNumber) to each new packet. The QS sorts the identifiers of the packets alive in XCaliber based on the priority of the packets, and it updates the sorting when the SPU core notifies any change on the status of a packet. The QS selects which packet identifiers will be provided next to the SPU. Again, the QS is a particular focus of the present application.

Register Transfer Unit (RTU) block 227, upon receiving a packet identifier (packetPage and packetNumber) from the QS, searches for an available context (229, Fig. 2) out of 8 contexts that XCaliber features in a preferred embodiment. For architectural and description purposes the contexts are considered a part of a broader Stream Processing Unit, although the contexts are shown in Fig. 2 as a separate unit 229.

In the case that no context is available, the RTU has the ability to notify the SPU about this event through a set of interrupts. In the case that a context is available, the RTU loads the packet identifier information and some selected fields of the header of the packet into the context, and afterwards it releases the context (which will at that time come under control of the SPU. The RTU accesses the header information of the packet through the SIU, since the packet could have been stored in the off-chip EPM.

Eventually a stream in the SPU core processes the context and notifies the QS of this fact. There are, in a preferred embodiment, eight streams in the DMS core. The QS then updates the status of the packet (to completed), and eventually this packet is selected for downloading (i.e. the

packet data of the corresponding packet is sent out of the XCaliber processor to one of the two external devices).

When a packet is selected for downloading, the QS sends the packetPage (among other information) to the PMMU block, which generates the corresponding line addresses to read the packet data from the LPM (in case the packet was stored in the on-chip local memory) or it will instruct the SIU to bring the packet from the external packet memory to the PMU. In any case, the lines of packet data read are buffered into the OutBuffer (OB) block, and from there sent out to the device through the 16-byte network output interface. This interface is independent of its input counterpart. The maximum aggregated bandwidth of this interface in a preferred embodiment is also 20 Gbps, 10Gbps per output device.

CommandUnit (CU) 213 receives commands sent by SPU 105. A command corresponds to a packet instruction, which are in many cases newly defined instructions, dispatched by the SPU core. These commands are divided into three independent types, and the PMU can execute one command per type per cycle (for a total of up to 3 commands per cycle). Commands can be load-like or store-like (depending on whether the PMU provides a response back to the SPU or not, respectively).

A large number of features of the PMU are configured by the SPU through memory-mapped configuration registers 221. Some such features have to be programmed at boot time, and the rest can be dynamically changed. For some of the latter, the SPU has to be running in a single-thread mode to properly program the functionality of the feature. The CU block manages the update of these configuration registers.

The PMU provides a mechanism to aid in flow control between ASIC 203 and XCaliber DMS processor 201. Two different interrupts are generated by the PMU to SPU 105 when LPM 219 or QS 211 are becoming



full. Software controls how much in advance the interrupt is generated before the corresponding structure becomes completely full. Software can also disable the generation of these interrupts.

5 LPM 219 is also memory mapped, and SPU 105 can access it through the conventional load/store mechanism. Both configuration registers 221 and LPM 219 have a starting address (base address) kept by SIU 107. Requests from SPU 105 to LPM 219 and the configuration space arrive to the PMU through SIU block 107. The SIU is also aware of the base address of the external packet memory.

### **In Buffer (IB)**

Packet data sent by an external device arrives to the PMU through the network input interface 205 at an average rate of 8 bytes every XCaliber core cycle in a preferred embodiment. IB block 207 of the PMU receives this data, buffers it, and provides it, in a FIFO-like fashion, to LPM 219 and in some cases also to the SIU (in case of a packet overflow, as explained elsewhere in this specification.

20 XCaliber DMS processor 201 can potentially send/receive packet data to/from up to 2 independent devices. Each device is tagged in SIU 107 with a device identifier, which is provided along with the packet data. When one device starts sending data from a packet, it will continue to send data from that very same packet until the end of the packet is reached or a bus error is detected by the SIU.

25 In a preferred embodiment the first byte of a packet always starts at byte 0 of the first 16 bytes sent of that packet. The first two bytes of the

packet specify the size in bytes of the packet (including these first two bytes). These two bytes are always appended by the SIU if the external device has not appended them. If byte  $k$  in the 16-byte chunk is a valid byte, bytes  $0..k-1$  are also valid bytes. This can be guaranteed since the first byte of a packet always starts at byte 0. Note that no valid bits are needed to validate each byte since a packet always starts at byte 0 of the 16-byte chunk, and the size of the packet is known up front (in the first two bytes). The network interface provides, at every core clock, a control bit specifying whether the 16-byte chunk contains, at least, one valid byte.

The valid data received from the network input interface is organized in buffer 207. This is an 8-entry buffer, each entry holding the 16-bytes of data plus the control bits associated to each chunk. PMMU 209 looks at the control bits in each entry and determines whether a new packet starts or to which of the (up to) two active packets the data belongs to, and it acts accordingly.

The 16-byte chunks in each of the entries in IB 207 are stored in LPM 219 or in the EPM (not shown). It is guaranteed by either the LPM controller or the SIU that the bandwidth to write into the packet memory will at least match the bandwidth of the incoming packet data, and that the writing of the incoming packet data into the packet memory will have higher priority over other accesses to the packet memory.

In some cases IB 207 may get full because PMMU 209 may be stalled, and therefore the LPM will not consume any more data of the IB until the stall is resolved. Whenever the IB gets full, a signal is sent to network input interface 205, which will retransmit the next 16-byte chunk as many times as needed until the IB accepts it. Thus, no packet data is lost due to the IB getting full.

## Out Buffer (OB)

Network output interface 215 also supports a total aggregated bandwidth of 20 Gbps (10Gbps per output device), as does the Input Interface. At 300 MHz XCaliber clock frequency, the network output interface accepts in average 8 bytes of data every XCaliber cycle from the OB block, and sends it to one of the two output devices. The network input and output interfaces are completely independent of each other.

Up to 2 packets (one per output device) can be simultaneously sent. The device to which the packet is sent does not need to correspond to the device that sent the packet in. The packet data to be sent out will come from either LPM 219 or the EPM (not shown).

For each of the two output devices connected at Network Out interface 215, PMMU 209 can have a packet ready to start being downloaded, a packet being downloaded, or no packet to download. Every cycle PMMU 209 selects the highest packet across both output devices and initiates the download of 16 bytes of data for that packet. Whenever the PMMU is downloading packet data from a packet to an output device, no data from a different packet will be downloaded to the same device until the current packet is completely downloaded.

The 16-byte chunks of packet data read from LPM 219 (along with some associated control information) are fed into one of the two 8-entry buffers (one per device identifier). The contents of the head of one of these buffers is provided to the network output interface whenever this interface requests it. When the head of both buffers is valid, the OB provides the data in a round robin fashion.

Differently than the network input interface, in the 16-byte chunk sent to the network output interface it can not be guaranteed that if a byte  $k$  is valid, then bytes  $0..k-1$  are valid as well. The reason for this is that when the packet is being sent out, it does not need to start at byte 0 of the 16-byte chunk in memory. Thus, for each 16-byte chunk of data that contains the start of the packet to be sent out, OB 217 needs to notify the network interface where the first valid byte of the chunk resides. Moreover, since the first two bytes of the packet contain the size of the packet in bytes, the network output interface has the information to figure out where the last valid byte of the packet resides within the last 16-byte chunk of data for that packet. Moreover, OB 217 also provides a control bit that informs SIU whether it needs to compute CRC for the packet, and if so, which type of CRC. This control bit is provided by PMMU 209 to OB 217.

### **Paging Memory Management Unit (PMMU)**

The packet memory address space is 16MB. Out of the 16MB, the XCaliber processor features 256KB on-chip. The rest (or a fraction) is implemented using external storage.

The packet memory address space can be mapped in the TLB of SPU 105 as user or kernel space, and as cachable or uncachable. In case it is mapped cachable, the packet memory space is cached (write-through) into an L1 data cache of SPU 105, but not into an L2 cache.

A goal of PMMU 209 is to store incoming packets (and SPU-generated packets as well) into the packet memory. In case a packet from the network input interface fits into LPM 219, PMMU 209 decides where to store it and generates the necessary write accesses to LPM 219; in case the packet from the network input interface is going to be stored in the EPM,

SPU 105 decides where in the EPM the packet needs to be stored and SIU 107 is in charge of storing the packet. In either case, the packet is consecutively stored and a packet identifier is created by PMMU 209 and sent to QS 211.

5           SPU 105 can configure LPM 219 so packets larger than a given size will never be stored in the LPM. Such packets, as well as packets that do not fit into the LPM because lack of space, are sent by PMMU 209 to the EPM through SIU 107. This is a mechanism called overflow and is configured by the SPU for the PMU to do so. If no overflow of packets is  
10           allowed, then the packet is dropped. In this case, PMMU 209 interrupts the SPU (again, if configured to do so).

### Uploading a packet into packet memory

Whenever there is valid data at the head of IB 205, the corresponding device identifier bit is used to determine to which packet (out of the two possible packets being received) the data belongs. When the network input interface starts sending data of a new packet with device identifier  $d$ , all the rest of the data will eventually arrive with that same device identifier  $d$  unless  
15           an error is notified by the network interface block. The network input interface can interleave data from two different device identifiers, but in a  
20           given cycle only data from one device is received by IB 207.

When a packet needs to be stored into LPM 219, PMMU block 209 generates all the write addresses and write strobes to LPM 219. If the  
25           packet needs to be stored into the EPM, SIU 107 generates them.

Fig. 3 is a diagram illustrating uploading of data into either LPM 219 or the EPM, which is shown in Fig. 3 as element 305, but not shown in Fig. 2. The write strobe to the LPM or EPM will not be generated unless the

header of the IB has valid data. Whenever the write strobe is generated, the 16-byte chunk of data at the head of the IB (which corresponds to a LPM line) is deleted from the IB and stored in the LPM or EPM. The device identifier bit of the head of the IB is used to select the correct write address out of the 2 address generators (one per input device).

In the current embodiment only one incoming packet can be simultaneously stored in the EPM by the SIU (i.e. only one overflow packet can be handled by the SIU at a time). Therefore, if a second packet that needs to be overflowed is sent by the network input interface, the data of this packet will be thrown away (i.e. the packet will be dropped).

### **A Two Byte Packet-Size Header**

The network input interface always appends two bytes to a packet received from the external device (unless this external device already does so, in which case the SIU will be programmed not to append them). This appended data indicates the size in bytes of the total packet, including the two appended bytes. Thus, the maximum size of a packet that is processed by the XCaliber DMS processor is 65535 bytes including the first two bytes.

The network output interface expects that, when the packet is returned by the PMU (if not dropped during its processing), the first two bytes also indicate the size of the processed packet. The size of the original packet can change (the packet can increase or shrink) as a result of processing performed by the XCaliber processor. Thus, if the processing results in increasing the size beyond 64K-1 bytes, it is the responsibility of software to chop the packet into two different smaller packets.

The PMU is more efficient when the priority of the packet being received is known up front. The third byte of the packet will be used for

priority purpose if the external device is capable of providing this information to the PMU. The software programs the PMU to either use the information in this byte or not, which is done through a boot-time configuration register named Log2InQueues.

5

### **Dropping a packet**

A packet completely stored in either LPM 219 or EPM 305 will be dropped only if SPU 105 sends an explicit command to the PMU to do so. No automatic dropping of packets already stored in the packet memory can occur. In other words, any dropping algorithm of packets received by the XCaliber DMS processor is implemented in software.

There are, however, several situations wherein the PMU may drop an incoming packet. These are (a) The packet does not fit in the LPM and the overflow of packets is disabled, (b) The total amount of bytes received for the packet is not the same as the number of bytes specified by the ASIC in the first two bytes of the ASIC-specific header, or (c) A transmission error has occurred between the external device and the network input interface block of the SIU. The PMMU block is notified about such an error.

For each of the cases (a), (b) and (c) above, an interrupt is generated to the SPU. The software can disable the generation of these interrupts using AutomaticPacketDropIntEnable, PacketErrorIntEnable on-the-fly configuration flags.

25

### **Virtual Pages**

An important process of PMMU 209 is to provide an efficient way to consecutively store packets into LPM 219 with as little memory

fragmentation as possible. The architecture in the preferred embodiment provides SPU 105 with a capability of grouping, as much as possible, packets of similar size in the same region of LPM 219. This reduces overall memory fragmentation.

5           To implement the low-fragmentation feature, LPM 219 is logically divided into 4 blocks of 64KB bytes each. Each block is divided into fixed *atomic pages* of 256 bytes. However, every block has *virtual* pages that range from 256 bytes up to 64KB, in power-of-2 increments. Software can enable/disable the different sizes of the virtual pages for each of the 4 blocks using an on-the-fly configuration register named VirtualPageEnable. This  
10           allows configuring some blocks to store packets of up to a certain size.

          The organization and features of the PMU assure that a packet of size  $s$  will never be stored in a block with a maximum virtual page size less than  $s$ . However, a block with a minimum virtual page size of  $r$  will accept packets of size smaller than  $r$ . This will usually be the case, for example, in which another block or blocks are configured to store these smaller packets, but is full.

          Software can get ownership of any of the four blocks of the LPM, which implies that the corresponding 64KB of memory will become software managed. A configuration flag exists per block (SoftwareOwned) for this  
20           purpose. The PMMU block will not store any incoming packet from the network input interface into a block in the LPM with the associated SoftwareOwned flag asserted. Similarly, the PMMU will not satisfy a GetSpace operation (described elsewhere) with memory of a block with its  
25           SoftwareOwned flag asserted. The PMMU, however, is able to download any packet stored by software in a software-owned block.

          The PMMU logic determines whether an incoming packet fits in any of the blocks of the LPM. If a packet fits, the PMMU decides in which of the



four blocks (since the packet may fit in more than one block), and the first and last atomic page that the packet will use in the selected block. The atomic pages are allocated for the incoming packet. When packet data stored in an atomic page has been safely sent out of the XCaliber processor through the network output interface, the corresponding space in the LPM can be de-allocated (i.e. made available for other incoming packets).

The EPM, like the LPM is also logically divided into atomic pages of 256 bytes. However, the PMMU does not maintain the allocation status of these pages. The allocation status of these pages is managed by software. Regardless of where the packet is stored, the PMMU generates an offset (in atomic pages) within the packet memory to where the first data of the packet is stored. This offset is named henceforth *packetPage*. Since the maximum size of the packet memory is 16MB, the *packetPage* is a 16-bit value.

As soon as the PMMU safely stores the packet in the LPM, or receives acknowledgement from SIU 107 that the last byte of the packet has been safely stored in the EPM, the *packetPage* created for that packet is sent to the QS. Operations of the QS are described in enabling detail below.

## **Generating the *packetPage* offset**

The PMMU always monitors the device identifier (*deviceId*) associated to the packet data at the head of the IB. If the *deviceId* is not currently active (i.e. the previous packet sent by that *deviceId* has been completely received), that indicates that the head of the IB contains the first data of a new packet. In this case, the first two bytes (*byte0* and *byte1* in the 16-byte chunk) specify the size of the packet in bytes. With the information of the size of the new incoming packet, the PMMU determines whether the

packet fits into LPM 219 and, if it does, in which of the four blocks it will be stored, plus the starting and ending atomic pages within that block.

The required throughput in the current embodiment of the PMMU to determine whether a packet fits in LPM 219 and, if so, which atomic pages are needed, is one packet every two cycles. One possible two-cycle implementation is as follows: (a) The determination happens in one cycle, and only one determination happens at a time (b) In the cycle following the determination, the atomic pages needed to store the packet are allocated and the new state (allocated/de-allocated) of the virtual pages are computed. In this cycle, no determination is allowed.

Fig. 4a is a diagram illustrating determination and allocation in parallel for local packet memory. The determination logic is performed in parallel for all of the four 64 KB blocks as shown.

Fig. 4b shows the state that needs to be maintained for each of the four 64KB blocks. This state, named *AllocationMatrix*, is recomputed every time one or more atomic pages are allocated or de-allocated, and it is an input for the determination logic. The *FitsVector* and *IndexVector* contain information computed from the *AllocationMatrix*.

$AllocationMatrix[VPSize][VPIndex]$  indicates whether virtual page number  $VPIndex$  of size  $VPSize$  in bytes is already allocated or not.  $FitsVector[VPSize]$  indicates whether the block has at least one non-allocated virtual page of size  $VPSize$ . If  $FitsVector[VPSize]$  is asserted,  $IndexVector[VPSize]$  vector contains the index of a non-allocated virtual page of size  $VPSize$ .

The SPU programs which virtual page sizes are enabled for each of the blocks. The  $EnableVector[VPSize]$  contains this information. This configuration is performed using the *VirtualPageEnable* on-the-fly configuration register. Note that the  $AllocationMatrix[[]]$ ,  $FitsVector[[]]$ ,

IndexVector[] and EnableVector[] are don't cares if the corresponding SoftwareOwned flag is asserted.

In this example the algorithm for the determination logic (for a packet of size  $s$  bytes) is as follows:

5

- 1) Fits logic: check, for each of the blocks, whether the packet fits in or not. If it fits, remember the virtual page size and the number of the first virtual page of that size.

For All Block  $j$  Do (can be done in parallel):

Fits[j] = ( $s \leq VPSize$ ) AND FitsVector[VPSize]

10

AND

Not SoftwareOwned

where  $VPSize$  is the smallest possible page size.

If (Fits[j])

$VPIndex[j] = IndexVector[VPSize]$

$MinVPS[j] = VPSize$

Else

$MinVPS[j] = \langle \text{Infinity} \rangle$

20

- 2) Block selection: the blocks with the smallest virtual page (enabled or not) that is able to fit the packet in are candidates. The block with the smallest enabled virtual page is selected.

If Fits[j] = FALSE for all  $j$  Then

$\langle \text{Packet does not fit in LPM} \rangle$

packetPage = OverflowAddress  $\gg 8$

25

Else

$C = \text{set of blocks with smallest } MinVPS \text{ AND}$

Fits[MinVPS]

$B = \text{block\# in } C \text{ with the smallest enabled virtual}$

page

(if more than one exists, pick the smallest block number)

If one or more blocks in  $C$  have virtual pages enabled

Then

Index = VPIndex[ $B$ ]

VPSize = MinVPS[ $B$ ]

NumAPs = ceil( $S/256$ )

packetPage = ( $B*64KB + \text{Index}*VPSize$ ) >>

8

Else

<Packet does not fit in LPM>

packetPage = OverflowAddress >> 8

If the packet fits in the LPM, the packetPage created is then the atomic page number within the LPM (there are up to 1K different atomic pages in the LPM) into which the first data of the packet is stored. If the packet does not fit, then the packetPage is the contents of the configuration register OverflowAddress right-shifted 8 bits. The packet overflow mechanism is described elsewhere in this specification, with a subheader "Packet overflow".

In the cycle following the determination of where the packet will be stored, the new values of the AllocationMatrix, FitsVector and IndexVector must be recomputed for the selected block. If FitsVector[ $VPSize$ ] is asserted, then IndexVector[ $VPSize$ ] is the index of the largest non-allocated virtual page possible for the corresponding virtual page size. If FitsVector[ $VPSize$ ] is de-asserted, then IndexVector[ $VPSize$ ] is undefined.

The number of atomic pages needed to store the packet is calculated (NumAPs) and the corresponding atomic pages are allocated. The allocation of the atomic pages for the selected block ( $B$ ) is done as follows:

1. The allocation status of the atomic pages in `AllocationMatrix[APsize][j..k]`,  $j$  being the first atomic page and  $k$  the last one ( $k-j+1 = \text{NumAPs}$ ), are set to allocated.
2. The allocation status of the virtual pages in `AllocationMatrix[r][s]` are updated following the mesh structure in Fig. 4b. (a  $2^{k+1}$ -byte virtual page will be allocated if any of the two  $2^k$ -byte virtual pages that it is composed of is allocated).

When the `packetPage` has been generated, it is sent to the QS for enqueueing. If the QS is full (very rare), it will not be able to accept the `packetPage` being provided by the PMMU. In this case, the PMMU will not be able to generate a new `packetPage` for the next new packet. This puts pressure on the IB, which might get full if the QS remains full for several cycles.

The PMMU block also sends the queue number into which the QS has to store the `packetPage`. How the PMMU generates this queue number is described below in sections specifically allocated to the QS.

### Page Allocation example

Figs. 5a and 5b illustrate an example of how atomic pages are allocated. For simplicity, the example assumes 2 blocks (0 and 1) of 2KB each, with an Atomic page size of 256 bytes, and both blocks have their `SoftwareOwned` flag de-asserted. Single and double cross-hatched areas represent allocated virtual pages (single cross-hatched pages correspond to the pages being allocated in the current cycle). The example shows how the pages get allocated for a sequence of packet sizes of 256, 512, 1K and 512

bytes. Note that, after this sequence, a 2K-byte packet, for example, will not fit in the example LPM.

Whenever the FitsVector[*VPSize*] is asserted, the IndexVector[*VPSize*] contains the largest non-allocated virtual page index for virtual page size *VPSize*. The reason for choosing the largest index is that the memory space is better utilized. This is shown in Figs. 6a and 6b, where two 256-byte packets are stored in a block. In scenario A, the 256-byte virtual page is randomly chosen, whereas in scenario B, the largest index is always chosen. As can be seen, the block in scenario A only allows two 512-byte virtual pages, whereas the block in scenario B allows three. Both, however, allow the same number of 256-byte packets since this is the smallest allocation unit. Note that the same effect is obtained by choosing the smallest virtual page index number all the time.

### Packet overflow

The only two reasons why a packet cannot be stored in the LPM are (a) that the size of the packet is larger than the maximum virtual page enabled across all 4 blocks; or (b) that the size of the packet is smaller than or equal to the maximum virtual page enabled but no space could be found in the LPM.

When a packet does not fit into the LPM, the PMMU will overflow the packet through the SIU into the EPM. To do so, the PMMU provides the initial address to the SIU (16-byte offset within the packet memory) to where the packet will be stored. This 20-bit address is obtained as follows:

- (a) The 16 MSB bits correspond to the 16 MSB bits of the OverflowAddress configuration register (i.e. the atomic page number within the packet

memory). (b) The 4 LSB bits correspond to the HeaderGrowthOffset configuration register. The packetPage value (which will be sent to the QS) for this overflowed packet is then the 16 MSB bits of the OverflowAddress configuration register.

5           If the on-the-fly configuration flag OverflowEnable is asserted, the PMMU will generate an OverflowStartedInt interrupt. When the OverflowStartedInt interrupt is generated, the size in bytes of the packet to overflow is written by the PMMU into the SPU-read-only configuration register SizeOfOverflowedPacket. At this point, the PMMU sets an internal  
10           lock flag that will prevent a new packet from overflowing. This lock flag is reset when the software writes into the on-the-fly configuration register OverflowAddress. If a packet needs to be overflowed but the lock flag is set, the packet will be dropped.

            With this mechanism, it is guaranteed that only one interrupt will be generated and serviced per packet that is overflowed. This also creates a platform for software to decide where the starting address into which the next packet that will be overflowed will be stored is visible to the interrupt service routine through the SizeOfOverflowedPacket register. In other  
15           words, software manages the EPM.

20           If software writes the OverflowAddress multiple times in between two OverflowStartedInt interrupts, the results are undefined. Moreover, if software sets the 16 MSB bits of OverflowAddress to 0..1023, results are also undefined since the first 1K atomic pages in the packet memory correspond to the LPM.

### 25           **Downloading a packet from packet memory**

            Eventually the SPU will complete the processing of a packet and will inform the QS of the fact. At this point the packet may be downloaded from

memory, either LPM or EPM, and sent, via the OB to one of the connected devices. Fig. 7 is a top-level schematic of the blocks of the XCaliber DMS processor involved in the downloading of a packet, and the elements in Fig. 7 are numbered the same as in Fig. 2. The downloading process may be followed in Fig. 7 with the aid of the following descriptions.

When QS 211 is informed that processing of a packet is complete, the QS marks this packet as completed and, a few cycles later (depending on the priority of the packet), the QS provides to PMMU 209 (as long as the PMMU has requested it) the following information regarding the packet:

- (a) the packetPage
- (b) the priority (cluster number from which it was extracted)
- (c) the tail growth/shrink information (described later in spec)
- (d) the outbound device identifier bit
- (e) the CRC type field (described later in spec)
- (f) the KeepSpace bit

The device identifier sent to PMMU block 209 is a 1-bit value that specifies the external device to which the packet will be sent. This outbound device identifier is provided by software to QS 211 as a 2-bit value.

If the packet was stored in LPM 219, PMMU 209 generates all of the (16-byte line) read addresses and read strobes to LPM 219. The read strobes are generated as soon as the read address is computed and there is enough space in OB 217 to buffer the line read from LPM 219. Buffer  $d$  in the OB is associated to device identifier  $d$ . This buffer may become full for either two reasons: (a) The external device  $d$  temporarily does not accept data from XCaliber; or (b) The rate of reading data from the OB is lower than the rate of writing data into it.

As soon as the packet data within an atomic page has all been downloaded and sent to the OB, that atomic page can be de-allocated. The



de-allocation of one or more atomic pages follows the same procedure as described above. However, no de-allocation of atomic pages occurs if the LPM bit is de-asserted. The KeepSpace bit is a don't care if the packet resides in EPM 701.

5           If the packet was stored in EPM 701, PMMU 209 provides to SIU  
107 the address within the EPM where the first byte of the packet resides.  
The SIU performs the downloading of the packet from the EPM. The SIU  
also monitors the buffer space in the corresponding buffer in OB 217 to  
determine whether it has space to write the 16-byte chunk read from EPM  
10 701. When the packet is fully downloaded, the SIU informs the PMMU of  
the fact so that the PMMU can download the next packet with the same  
device identifier.

When two packets (one per device) are being simultaneously sent, data  
from the packet with highest priority is read out of the memory first. This  
preemption can happen at a 16-byte boundary or when the packet finishes its  
transmission. If both packets have the same priority (provided by the QS), a  
round-robin method is used to select the packet from which data will be  
downloaded next. This selection logic also takes into account how full the  
two buffers in the OB are. If buffer  $d$  is full, for example, no packet with a  
device identifier  $d$  will be selected in the PMMU for downloading the next  
20 16-byte chunk of data.

When a packet starts to be downloaded from the packet memory (local  
or external), the PMMU knows where the first valid byte of the packet  
resides. However, the packet's size is not known until the first line (or the  
25 first two lines in some cases) of packet data is read from the packet memory,  
since the size of the packet resides in the first two bytes of the packet data.  
Therefore, the processing of downloading a packet first generates the

necessary line addresses to determine the size of the packet, and then, if needed, generates the rest of the accesses.

This logic takes into account that the first two bytes that specify the size of the packet can reside in any position in the 16-byte line of data. A particular case is when the first two bytes span two consecutive lines (which will occur when the first byte is the 16th byte of a line, and second byte is the 1<sup>st</sup> byte of next line.

As soon as the PMMU finishes downloading a packet (all the data of that packet has been read from packet memory and sent to OB), the PMMU notifies the QS of this event. The QS then invalidates the corresponding packet from its queuing system.

When a packet starts to be downloaded, it cannot be preempted, i.e. the packet will finish its transmission. Other packets that become ready to be downloaded with the same outbound device identifier while the previous packet is being transmitted cannot be transmitted until the previous packet is fully transmitted.

### **Packet growth/shrink**

As a result of processing a packet, the size of a network packet can grow, shrink or remain the same size. If the size varies, the SPU has to write the new size of the packet in the same first two bytes of the packet. The phenomenon of packet growth and shrink is illustrated in Fig. 8.

Both the header and the tail of the packet can grow or shrink. When a packet grows, the added data can overwrite the data of another packet that may have been stored right above the packet experiencing header growth, or that was stored right below in the case of tail growth. To avoid this problem

the PMU can be configured so that an empty space is allocated at the front and at the end of every packet when it is stored in the packet memory.

These empty spaces are specified with HeaderGrowthOffset and TailGrowthOffset boot-time configuration registers, respectively, and their granularity is 16 bytes. The maximum HeaderGrowthOffset is 240 bytes (15 16-byte chunks), and the maximum TailGrowthOffset is 1008 bytes (63 16-byte chunks). The minimum in both cases is 0 bytes. Note that these growth offsets apply to all incoming packets, that is, there is no mechanism to apply different growth offsets to different packets.

When the PMMU searches for space in the LPM, it will look for contiguous space of  $\text{Size}(\text{packet}) + ((\text{HeaderGrowthOffset} + \text{TailGrowthOffset}) \ll 4)$ . Thus, the first byte of the packet (first byte of the ASIC-specific header) will really start at offset  $((\text{packetPage} \ll 8) + (\text{HeaderGrowthOffset} \ll 4))$  within the packet memory.

The software knows what the default offsets are, and, therefore, knows how much the packet can safely grow at both the head and the tail. In case the packet needs to grow more than the maximum offsets, the software has to explicitly move the packet to a new location in the packet memory. The steps to do this are as follows:

- 1) The software requests the PMU for a chunk of contiguous space of the new size. The PMU will return a new packetPage that identifies (points to) this new space.
- 2) The software writes the data into the new memory space.
- 3) The software renames the old packetPage with the new packetPage.
- 4) The software requests the PMU to de-allocate the space associated to the old packetPage.

In the case of header growth or shrinkage, the packet data will no longer start at  $((\text{packetPage} \ll 8) + (\text{HeaderGrowthOffset} \ll 4))$ . The new starting location is provided to the PMU with a special instruction executed by the SPU when the processing of the packet is completed. This information is provided to the PMMU by the QS block.

### Time stamp

The QS block of the PMU (described in detail in a following section) guarantees the order of the incoming packets by keeping the packetPage identifiers of the packets in process in the XCaliber processor in FIFO-like queues. However, software may break this ordering by explicitly extracting identifiers from the QS, and inserting them at the tail of any of the queues.

To help software in guaranteeing the relative order of packets, the PMU can be configured to time stamp every packet that arrives to the PMMU block using an on-the-fly configuration flag TimeStampEnabled. The time stamp is an 8-byte value, obtained from a 64-bit counter that is incremented every core clock cycle.

When the time stamp feature is on, the PMMU appends the 8-byte time stamp value in front of each packet, and the time stamp is stripped off when the packet is sent to the network output interface. The time stamp value always occupies the 8 MSB bytes of the  $(k-1)$ th 16-byte chunk of the packet memory, where  $k$  is the 16-byte line offset where the data of the packet starts ( $k > 0$ ). In the case that HeaderGrowthOffset is 0, the time stamp value will not be appended, even if TimeStampEnabled is asserted.

The full 64-bit time counter value is provided to software through a read-only configuration register (TimeCounter).

## Software operations on the PMMU

5                Software has access to the PMMU to request or free a chunk of  
contiguous space. In particular, there are two operations that software can  
perform on the PMMU. Firstly the software, through an operation  
GetSpace(*size*), may try to find a contiguous space in the LPM for *size*  
bytes. The PMU replies with the atomic page number where the contiguous  
10              space that has been found starts (i.e. the *packetPage*), and a success bit. If  
the PMU was able to find space, the success bit is set to '1', otherwise it is  
set to '0'. GetSpace will not be satisfied with memory of a block that has its  
SoftwareOwned configuration bit asserted. Thus, software explicitly  
manages the memory space of software-owned LPM blocks.

                The PMMU allocates the atomic pages needed for the requested  
space. The EnableVector set of bits used in the allocation of atomic pages  
for incoming packets is a don't care for the GetSpace operation. In other  
words, as long as sufficient consecutive non-allocated atomic pages exist in a  
particular block to cover *size* bytes, the GetSpace(*size*) operation will  
20              succeed even if all the virtual pages in that block are disabled.  
Moreover, among non-software-owned blocks, a GetSpace operation will be  
served first using a block that has all its virtual pages disabled. If more than  
such a block exists, the smallest block number is chosen. If *size* is 0,  
GetSpace(*size*) returns '0'.

25              The second operation software can perform on the PMMU is  
FreeSpace(*packetPage*). In this operation the PMU de-allocates atomic  
pages that were previously allocated (starting at *packetPage*). This space  
might have been either automatically allocated by the PMMU as a result of

an incoming packet, or as a result of a GetSpace command. FreeSpace does not return any result to the software. A FreeSpace operation on a block with its SoftwareOwned bit asserted is disregarded (nothing is done and no result will be provided to the SPU).

5

### Local Packet Memory

Local Packet Memory (LPM), illustrated as element 219 in FIGS. 2 and 7, has in the instant embodiment a size of 256KB, 16-byte line width with byte enables, 2 banks (even/odd), one Read and one Write port per bank, is fully pipelined, and has one cycle latency

The LPM in packet processing receives read and write requests from both the PMMU and the SIU. An LPM controller guarantees that requests from the PMMU have the highest priority. The PMMU reads at most one packet while writing another one. The LPM controller guarantees that the PMMU will always have dedicated ports to the LPM.

Malicious software could read/write the same data that is being written/read by the PMMU. Thus, there is no guarantee that the read and write accesses in the same cycle are performed to different 16-byte line addresses.

A request to the LPM is defined in this example as a single access (either read or write) of 16-bytes. The SIU generates several requests for a masked load or store, which are new instructions known to the inventors and the subject of at least one separate patent application. Therefore, a masked load/store operation can be stalled in the middle of these multiple requests if the highest priority PMMU access needs the same port.

10

11  
12  
13  
14  
15  
16  
17  
18  
19

20

25

When the PMMU reads or writes, the byte enable signals are assumed to be set (i.e. all 16 bytes in the line are either read or written). When the SIU drives the reads or writes, the byte enable signals are meaningful and are provided by the SIU.

When the SPU reads a single byte/word in the LPM, the SIU reads the corresponding 16-byte line and performs the extraction and right alignment of the desired byte/word. When the SPU writes a single byte/word, the SIU generates a 16-byte line with the byte/word in the correct location, plus the valid bytes signals.

### **Prioritization among operations**

The PMMU may receive up to three requests from three different sources (IB, QS and software) to perform operations. For example, requests may come from the IB and/or Software: to perform a search for a contiguous chunk of space, to allocate the corresponding atomic page sizes and to provide the generated packetPage. Requests may also come from the QS and/or Software to perform the de-allocation of the atomic pages associated to a given packetPage.

It is required that the first of these operations takes no more than 2 cycles, and the second no more than one. The PMMU executes only one operation at a time. From highest to lowest, the PMMU block will give priority to requests from: IB, QS and Software.

### **Early full-PMMU detection**

The PMU implements a mechanism to aid in flow control between any external device and the XCaliber processor. Part of this mechanism is to

detect that the LPM is becoming full and, in this case, a NoMorePagesOfXsizeInt interrupt is generated to the SPU. The EPM is software controlled and, therefore, its state is not maintained by the PMMU hardware.

The software can enable the NoMorePagesOfXsizeInt interrupt by specifying a virtual page size  $s$ . Whenever the PMMU detects that no more available virtual pages of that size are available (i.e. FitsVector[ $s$ ] is de-asserted for all the blocks), the interrupt is generated. The larger the virtual page size selected, the sooner the interrupt will be generated. The size of the virtual page will be indicated with a 4-bit value (0:256 bytes, 1:512 bytes, ..., 8:64KB) in an on-the-fly configuration register IntIfNoMoreThanXsizePages. When this value is greater than 8, the interrupt is never generated.

If the smallest virtual page size is selected (256 bytes), the NoMorePagesOfXsizeInt interrupt is generated when the LPM is completely full (i.e. no more packets are accepted, not even a 1-byte packet).

In general, if the IntIfNoMoreThanXsizePages is  $X$ , the soonest the interrupt will be generated is when the local packet memory is  $(100/2^X)\%$  full. Note that, because of the atomic pages being 256 bytes, the LPM could become full with only 3 K-bytes of packet data (3 byte per packet, each packet using an atomic page).

### Packet size mismatch

The PMMU keeps track of how many bytes are being uploaded into the LPM or EPM. If this size is different from the size specified in the first two bytes, a PacketErrorInt interrupt is generated to the SPU. In this case



the packet with the mismatch packet size is dropped (the already allocated atomic pages will be de-allocated and no packetPage will be created). No AutomaticDropInt interrupt is generated in this case. If the actual size is more than the size specified in the first two bytes, the remaining packet data being received from the ASIC is gracefully discarded.

When a packet size mismatch is detected on an inbound device identifier D ( $D = 0,1$ ), the following packets received from that same device identifier are dropped until software writes (any value) into a ClearErrorD configuration register.

### Bus Error Recovering

Faulty packet data can arrive to or leave the PMU due to external bus errors. In particular the network input interface may notify that the 16-byte chunk of data sent in has a bus error, or the SIU may notify that the 16-byte chunk of data downloaded from EPM has a bus error. In both cases, the PMMU generates the PacketErrorInt interrupt to notify the SPU about this event. No other information is provided to the SPU.

Note that if an error is generated within the LPM, it will not be detected since no error detection mechanism is implemented in this on-chip memory. Whenever a bus error arises, no more data of the affected packet will be received by the PMU. This is done by the SIU in both cases. For the first case the PMMU needs to de-allocate the already allocated atomic pages used for the packet data received previous to the error event.

When a bus error is detected on an inbound device identifier D ( $D = 0,1$ ), the following packets received from that same device identifier are

dropped until software writes (any value) into a `ClearErrorD` ( $D=0,1$ ) configuration register.

## 5      **Queuing System (QS)**

The queueing system (QS) in the PMU of the XCaliber processor has functions of holding packet identifiers and the state of the packets currently in-process in the XCaliber processor, keeping packets sorted by their default or software-provided priority, selecting the packets that need to be pre-loaded (in the background) into one of the available contexts, and selecting those processed packets that are ready to be sent out to an external device.

Fig. 9 is a block diagram showing the high-level communication between the QS and other blocks in the PMU and SPU. When the PMMU creates a `packetPage`, it is sent to the QS along with a queue number and the device identifier. The QS enqueues that `packetPage` in the corresponding queue and associates a number (`packetNumber`) to that packet. Eventually, the packet is selected and provided to the RTU, which loads the `packetPage`, `packetNumber` and selected fields of the packet header into an available context. Eventually the SPU processes that context and communicates to the PMU, among other information, when the processing of the packet is completed or the packet has been dropped. For this communication, the SPU provides the `packetNumber` as the packet identifier. The QS marks that packet as completed (in the first case) and the packet is eventually selected for downloading from packet memory.

It is a requirement in the instant embodiment (and highly desirable) that packets of the same flow (same source and destination) need to be sent out to the external device in the same order as they arrived to the XCaliber

processor (unless software explicitly breaks this ordering). When the SPU begins to process a packet the flow is not known. Keeping track of the ordering of packets within a flow is a costly task because of the amount of processing needed and because the number of active flows can be very large, depending on the application. Thus, the order within a flow is usually kept track by using aggregated-flow queues. In an aggregated-flow queue, packet identifiers from different flows are treated as from the same flow for ordering purposes.

The QS offloads the costly task of maintaining aggregated-flow queues by doing it in hardware and in the background. Up to 32 aggregated-flow queues can be maintained in the current embodiment, and each of these queues has an implicit priority. Software can enqueue a packetPage in any of the up to 32 queues, and can move a packetPage identifier from one queue to another (for example, when the priority of that packet is discovered by the software). It is expected that software, if needed, will enqueue all the packetPage identifiers of the packets that belong to the same flow into the same queue. Otherwise, a drop in the performance of the network might occur, since packets will be sent out of order within the same flow. Without software intervention, the QS guarantees the per-flow order of arrival.

### Generic Queue

The QS implements a set of up to 32 FIFO-like queues, which are numbered, in the case of 32 queues, from 0 to 31. Each queue can have up to 256 entries. The addition of all the entries of all the queues, however, cannot exceed 256. Thus, queue sizes are dynamic. A queue entry

corresponds to a packetPage identifier plus some other information. Up to 256 packets are therefore allowed to be in process at any given time in the XCaliber processor. This maximum number is not visible to software.

Whenever the QS enqueues a packetPage, a number (packetNumber) from 0 to 255 is assigned to the packetPage. This number is provided to the software along with the packetPage value. When the software wants to perform an operation on the QS, it provides the packetNumber identifier. This identifier is used by the QS to locate the packetPage (and other information associated to the corresponding packet) in and among its queues.

Software is aware that the maximum number of queues in the XCaliber processor is 32. Queues are disabled unless used. That is, the software does not need to decide how many queues it needs up front. A queue becomes enabled when at least one packet is in residence in that queue.

Several packet identifiers from different queues can become candidates for a particular operation to be performed. Therefore, some prioritization mechanism must exist to select the packet identifier to which an operation will be applied first. Software can configure (on-the-fly) the relative priority among the queues using an "on-the-fly" configuration register PriorityClusters. This is a 3-bit value that specifies how the different queues are grouped in clusters. Each cluster has associated a priority (the higher the cluster number, the higher the priority). The six different modes in the instant embodiment into which the QS can be configured are shown in the table of Fig. 10.

The first column of Fig. 10 is the value in the "on-the-fly" configuration register PriorityClusters. Software controls this number, which defines the QS configuration. For example, for PriorityClusters = 2, the QS is configured into four clusters, with eight queues per cluster. The

first of the four clusters will have queues 0 through 7, the second cluster will have queues 8-15, the third clusters 16 through 23, and the last of the four clusters has queues 24 through 31.

Queues within a cluster are treated fairly in a round robin fashion.

Clusters are treated in a strict priority fashion. Thus, the only mode that guarantees no starvation of any queue is when PriorityClusters is 0, meaning one cluster of 32 queues.

### Inserting a packetPage/deviceId into the QS

Fig. 11 is a diagram illustrating the generic architecture of QS 211 of Figs. 2 and 7 in the instant embodiment. Insertion of packetPages and DeviceId information is shown as arrows directed toward the individual queues (in this case 32 queues). The information may be inserted from three possible sources, these being the PMMU, the SPU and re-insertion from the QS. There exists priority logic, illustrated by function element 1101, for the case in which two or more sources have a packetPage ready to be inserted into the QS. In the instant embodiment the priority is, in descending priority order, the PMMU, the QS, and the SPU (software).

Regarding insertion of packets from the SPU (software), the software can create packets on its own. To do so, it first requests a consecutive chunk of free space of a given size (see the SPU documentation) from the PMU, and the PMU returns a packetPage in case the space is found. The software needs to explicitly insert that packetPage for the packet to be eventually sent out. When the QS inserts this packetPage, the packetNumber created is sent to the SPU. Software requests an insertion through the Command Unit (see Fig. 2).

In the case of insertion from the QS, an entry residing at the head of a queue may be moved to the tail of another queue. This operation is shown as selection function 1103.

In the case of insertion from the PMU, when a packet arrives to the XCaliber processor, the PMMU assigns a packetPage to the packet, which is sent to the QS as soon as the corresponding packet is safely stored in packet memory.

An exemplary entry in a queue is illustrated as element 1105, and has the following fields: Valid (1) validates the entry. PacketPage (16) is the first atomic page number in memory used by the packet. NextQueue (5) may be different from the queue number the entry currently belongs to, and if so, this number indicates the queue into which the packetPage needs to be inserted next when the entry reaches the head of the queue. Delta (10) contains the number of bytes that the header of the packet has either grown or shrunk. This value is coded in 2's complement. Completed (1) is a single bit that indicates whether software has finished the processing of the corresponding packet. DeviceId (2) is the device identifier associated to the packet. Before a *Complete* operation is performed on the packet (described below) the DeviceId field contains the device identifier of the external device that sent the packet in. After the *Complete* operation, this field contains the device identifier of the device to which the packet will be sent. Active (1) is a single bit that indicates whether the associated packet is currently being processed by the SPU. CRCtype (2) indicates to the network output interface which type of CRC, if any, needs to be computed for the packet. Before the Complete operation is performed on the packet, this field is 0. KeepSpace (1) specifies whether the atomic pages that the packet occupies in the LPM will be de-allocated (KeepSpace de-asserted) by the PMMU or

not (KeepSpace asserted). If the packet resides in EPM this bit is disregarded by the PMMU.

The QS needs to know the number of the queue to which the packetPage will be inserted. When software inserts the packetPage, the queue number is explicitly provided by an XStream packet instruction, which is a function of the SPU, described elsewhere in this specification. If the packetPage is inserted by the QS itself, the queue number is the value of the NextQueue field of the entry where the packetPage resides.

When a packetPage is inserted by the PMMU, the queue number depends on how the software has configured (at boot time) the Log2InputQueues configuration register. If Log2InputQueues is set to 0, all the packetPages for the incoming packets will be enqueued in the same queue, which is specified by the on-the-fly configuration register FirstInputQueue. If Log2InputQueues is set to  $k$  ( $1 \leq k \leq 5$ ), then the  $k$  MSB bits of the 3rd byte of the packet determine the queue number. Thus an external device (or the network input interface block of the SIU) can assign up to 256 priorities for each of the packets sent into the PMU. The QS maps those 256 priorities into  $2^k$ , and uses queue numbers FirstInputQueue to FirstInputQueue+ $2^k$ -1 to insert the packetPages and deviceId information of the incoming packets.

It is expected that an external device will send the same 5 MSB bits in the 3<sup>rd</sup> byte for all packets in the same flow. Otherwise, a drop in the performance of the network might occur, since packets may be sent back to the external device out-of-order within the same flow. Software is aware of whether or not the external device (or SIU) can provide the information of the priority of the packet in the 3<sup>rd</sup> byte.

When packetPage  $p$  is inserted into queue  $q$ , the PacketPage field of the entry to be used is set to  $p$  and the Valid field to '1'. The value for the

other fields depend on the source of the insertion. If the source is software (SPU), Completed is '0'; NextQueue is provided by SPU; DeviceId is '0'; Active is '1'; CRCtype is 0; KeepSpace is 0, and Probed is 0.

If the source is the QS, the remaining fields are assigned the value they have in the entry in which the to-be-inserted packetPage currently resides. If the source is the PMMU, Completed is '0', NextQueue is  $q$ , DeviceId is the device identifier of the external device that sent the packet into XCaliber, Active is '0', CRCtype is 0, KeepSpace is 0, and Probed is 0.

### Monitoring logic

The QS monitors entries into all of the queues to detect certain conditions and to perform the corresponding operation, such as to re-enqueue an entry, to send a packetPage (plus some other information) to the PMMU for downloading, or to send a packetPage (plus some other information) to the RTU.

All detections take place in a single cycle and they are done in parallel.

### Re-enqueueing an entry

The QS monitors all the head entities of the queues to determine whether a packet needs to be moved to another queue. Candidate entries to be re-enqueued need to be valid, be at the head of a queue, and have the NextQueue field value different from the queue number of the queue in which the packet currently resides.

If more than one candidate exists for re-enqueueing, the chosen entry will be selected following a priority scheme described later in this specification.



### **Sending an entry to the PMMU for downloading**

The QS monitors all the head entities of the queues to determine whether a packet needs to be downloaded from the packet memory. This operation is 1102 in Fig. 11. The candidate entries to be sent out of XCaliber need to be valid, be at the head of the queue, have the NextQueue field value the same as the queue number of the queue in which the packet currently resides, and have the Completed flag asserted and the Active flag de-asserted. Moreover the QS needs to guarantee that no pending reads or writes exist from the same context that has issued the download command to the QS.

If more than one candidate exists for downloading, the chosen entry will be selected following a priority scheme described later in this specification.

A selected candidate will only be sent to the PMMU if the PMMU requested it. If the candidate was requested, the selected packetPage, along with the cluster number from which it is extracted, the tail growth/shrink, the outbound device identifier bit, the CRCtype and the KeepSpace bits are sent to the PMMU.

Fig. 12 is a table indicating coding of the Deviceid field. If the Deviceid field is 0, then the Outbound Device Identifier is the same as the Inbound Device Identifier, and so on as per the table.

When an entry is sent to the PMMU, the entry is marked as “being transmitted” and it is extracted from the queuing system (so that it does not block other packets that are ready to be transmitted and go to a different outbound device identifier). However, the entry is not invalidated until the PMMU notifies that the corresponding packet has been completely

downloaded. Thus, probe-type operations on this entry will be treated as valid, i.e. as still residing in the XCaliber processor.

## 5      **Reincarnation effect**

As described above, the QS assigns a packetNumber from 0 to 255 (256 numbers in total) to each packet that comes into XCaliber and is inserted into a queue. This is done by maintaining a table of 256 entries into which packet identifiers are inserted. At this time the Valid bit in the packet identifier is also asserted. Because the overall numbers of packets dealt with by XCaliber far exceeds 256, packet numbers, of course, have to be reused throughout the running of the XCaliber processor. Therefore, when packets are selected for downloading, at some point the packetNumber is no longer associated with a valid packet in process, and the number may be reused.

As long as a packet is valid in XCaliber it is associated with the packetNumber originally assigned. The usual way in which a packetNumber becomes available to be reused is that a packet is sent by the QS to the RTU for preloading in a context prior to processing. Then when the packet is fully processed and fully downloaded from memory, the packet identifier in the table associating packetNumbers is marked Invalid by manipulating the Valid bit (see Fig. 11 and the text accompanying).

In usual operation the system thus far described is perfectly adequate. It has been discovered by the inventors, however, that there are some situations in which the Active and Valid bits are not sufficient to avoid contention between streams. One of these situations has to do with a clean-up process, sometimes termed *garbage collection*, in which software monitors all packet numbers to determine when packets have remained in the

system too long, and discards packets under certain conditions, freeing space in the system for newly-arriving packets.

In these special operations, like *garbage collection*, a stream must gain ownership of a packet, and assure that the operation it is to perform on the packet actually gets performed on the correct packet. As software probes packets, however, and before action may be taken, because there are several streams operating, and because the *normal* operation of the system may also send packets to the RTU, for example, it is perfectly possible in these special operations that a packet probed may be selected and effected by another stream before the special operation is completed. A packet, for example, may be sent to the RTU, processed, and downloaded, and a new packet may then be assigned to the packetNumber, and the new packet may even be stored at exactly the same packetPage as the original packet. There is a danger, then, that the special operations, such as discarding a packet in the garbage collection process, may discard a new and perfectly valid packet, instead of the packet originally selected to be discarded. This, of course, is just one of potentially many such special operations that might lead to trouble.

Considering the above, the inventors have provided a mechanism for assuring that, given two different absolute points in time, time  $s$  and time  $r$ , for example, that a valid packetNumber at time  $s$  and the same packetNumber at time  $r$ , still is associated to the same packet. A simple probe operation is not enough, because at some time after  $s$  and before time  $r$  the associated packet may be downloaded, and another (and different) packet may have arrived, been stored in exactly the same memory location as the previous packet, and been assigned the same packetNumber as the downloaded packet.

The mechanism implemented in XCaliber to ensure packetNumber association with a specific packet at different times includes a probe bit in the packet identifier. When a first stream, performing a process such as garbage collection, probes a packet, a special command, called Probe&Set is used. Probe&Set sets (asserts) the probe bit, and the usual information is returned, such as the value for the Valid bit, the Active bit, the packetPage address, and the old value of the probe bit. The first stream then executes a Conditional Activate instruction, described elsewhere in this specification, to gain ownership of the packet. Also, when the queuing system executes this Conditional Activate instruction it asserts the active bit of the packet. Now, at any time after the probe bit is set by the first stream, when a second stream at a later time probes the same packet, the asserted probe bit indicates that the first stream intends to gain control of this packet. The second stream now knows to leave this packet alone. This probe bit is de-asserted when a packet enters the XCaliber processor and a new (non-valid) number is assigned.

### **Sending an entry to the RTU**

The RTU uploads in the SPU background to the XCaliber processor some fields of the headers of packets that have arrived, and have been completely stored into packet memory. This uploading of the header of a packet in the background may occur multiple times for the same packet. The QS keeps track of which packets need to be sent to the RTU. The selection operation is illustrated in Fig. 11 as 1104.

Whenever the RTU has chosen a context to pre-load a packet, it notifies the QS that the corresponding packet is no longer an inactive packet. The QS then marks the packet as active.

Candidate entries to be sent to the RTU need to be valid, to be the oldest entry with the Active and Completed bits de-asserted, to have the NextQueue field value the same as the queue number of the queue in which the packet currently resides, and to conform to a limitation that no more than a certain number of packets in the queue in which the candidate resides are currently being processed in the SPU. More detail regarding this limitation is provided later in this specification. When an entry is sent to the RTU for pre-loading, the corresponding Active bit is asserted.

A queue can have entries with packet identifiers that already have been presented to the RTU and entries that still have not. Every queue has a pointer (NextPacketForRTU) that points to the oldest entry within that queue that needs to be sent to the RTU. Within a queue, packet identifiers are sent to the RTU in the same order they were inserted in the queue.

The candidate packet identifiers to be sent to the RTU are those pointed to by the different NextPacketForRTU pointers associated with the queues. However, some of these pointers might point to a non-existent entry (for example, when the queue is empty or when all the entries have already been sent to the RTU). The hardware that keeps track of the state of each of the queues determines these conditions. Besides being a valid entry pointed to by a NextPacketForRTU pointer, the candidate entry needs to have associated with it an RTU priority (described later in this specification) currently not being used by another entry in the RTU. If more than a single candidate exists, the chosen entry is selected following a priority scheme described later in this specification.

As opposed to the case in which an entry is sent to the PMMU for downloading, an entry sent to the RTU is not extracted from its queue. Instead, the corresponding NextPacketForRTU pointer is updated, and the corresponding Active bit is asserted.

The QS sends entries to an 8-entry table in the RTU block as long as the entry is a valid candidate and the corresponding slot in the RTU table is empty. The RTU will accept, at most, 8 entries, one per each interrupt that the RTU may generate to the SPU.

5       The QS maps the priority of the entry (given by the queue number where it resides) that it wants to send to the RTU into one of the 8 priorities handled by the RTU (RTU priorities). This mapping is shown in the table of Fig. 13, and it depends on the number of clusters into which the different queues are grouped (configuration register PriorityClusters) and the queue  
10       number in which the entry resides.

15       The RTU has a table of 8 entries, one for each RTU priority. Every entry contains a packet identifier (packetPage, packetNumber, queue#) and a Valid bit that validates it. The RTU always accepts a packet identifier of RTU priority  $p$  if the corresponding Valid bit in entry  $p$  of that table is de-asserted. When the RTU receives a packet identifier of RTU priority  $p$  from the QS, the Valid bit of entry  $p$  in the table is asserted, and the packet identifier is stored. At that time the QS can update the corresponding  
20       NextPacketForRTU pointer.

## 20       **Limiting the packets sent within a queue**

25       Software can limit the number of packets that can be active (i.e. being processed by any of the streams in the SPU) on a per-queue basis. This is achieved through a MaxActivePackets on-the-fly configuration register, which specifies, for each queue, a value between 1 and 256 that corresponds to the maximum number of packets, within that queue, that can be being processed by any stream.

The QS maintains a counter for each queue  $q$  which keeps track of the current number of packets active for queue  $q$ . This counter is incremented whenever a packet identifier is sent from queue  $q$  to the RTU, a Move operation moves a packet into queue  $q$ , or an Insert operation inserts a packet identifier into queue  $q$ ; and decremented when any one the following operations are performed in any valid entry in queue  $q$ : a Complete operation, an Extract operation, a Move operation that moves the entry to a different queue, or a MoveAndReactivate operation that moves the entry to any queue (even to the same queue). Move, MoveAndReactivate, Insert, Complete and Extract are operations described elsewhere in this specification.

Whenever the value of the counter for queue  $q$  is equal to or greater than the corresponding maximum value specified in the MaxActivePackets configuration register, no entry from queue  $q$  is allowed to be sent to the RTU. The value of the counter could be greater since software can change the MaxActivePackets configuration register for a queue to a value lower than the counter value at the time of the change, and a queue can receive a burst of moves and inserts.

### Software operations on the QS

Software executes several instructions that affect the QS. The following is a list of all operations that can be generated to the QS as a result of the dispatch by the SPU core of an XStream packet instruction:

**Insert( $p, q$ ):** the packetPage  $p$  is inserted into queue  $q$ . A '1' will be returned to the SPU if the insertion was successful, and a '0' if not. The

insertion will be unsuccessful only when no entries are available (i.e. when all the 256 entries are valid).

**Move( $n, q$ ):** asserts to  $q$  the NextQueue field of the entry in which packetNumber  $n$  resides.

5           **MoveAndReactivate( $n, q$ ):** asserts to  $q$  the NextQueue field of the entry in which packetNumber  $n$  resides; de-asserts the Active bit.

**Complete( $n, d, e$ ):** asserts the Completed flag, the Delta field to  $d$  and the deviceId field to  $e$  of the entry in which packetNumber  $n$  resides. De-asserts the Active bit and de-asserts the KeepSpace bit.

10           **CompleteAndKeepSpace( $n, d, e$ ):** same as Complete() but it asserts the KeepSpace bit.

**Extract( $n$ ):** resets the Valid flag of the entry in which packetNumber  $n$  resides.

15           **Replace( $n, p$ ):** the PacketPage field of the entry in which packetNumber  $n$  resides is set to packetPage  $p$ .

**Probe( $n$ ):** the information whether the packetNumber  $n$  exists in the QS or not is returned to the software. In case it exists, it returns the PacketPage, Completed, NextQueue, DeviceId, CRCtype, Active, KeepSpace and Probed fields.

20           **ConditionalActivate( $n$ ):** returns a '1' if the packetNumber  $n$  is valid, Probed is asserted, Active is de-asserted, and the packet is not being transmitted. In this case, the Active bit is asserted.

25           The QS queries the RTU to determine whether the packet identifier of the packet to be potentially activated is in the RTU table, waiting to be preloaded, or being preloaded. If the packet identifier is in the table, the RTU invalidates it. If the query happens simultaneously with the start of preloading of that packet, the QS does not activate the packet.



**ProbeAndSet(*n*):** same as Probe() but it asserts the Probed bit (the returned Probed bit is the old Probed bit).

**Probe(*q*):** provides the size (i.e. number of valid entries) in queue *q*.

5           A Move(), MoveAndReactivate(), Complete(),  
CompleteAndKeepSpace(), Extract() and Replace() on an invalid (i.e. non-existing) packetNumber is disregarded (no interrupt is generated).

10           A Move, MoveAndReactivate, Complete, CompleteAndKeepSpace,  
Extract and Replace on a valid packetNumber with the Active bit de-asserted should not happen (guaranteed by software). If it happens, results are undefined. Only the Insert, Probe, ProbeAndSet and ConditionalActivate operations reply back to the SPU.

15           If software issues two move-like operations to the PMU that affect the same packet, results are undefined, since there is no guarantee that the moves will happen as software specified.

            Fig. 14 is a table showing allowed combinations of Active, Completed, and Probed bits for a valid packet.

## 20           **Basic operations**

            To support the software operations and the monitoring logic, the QS implements the following basic operations:

- 25           1. Enqueue an entry at the tail of a queue.
2. Dequeue an entry from the queue in which it resides.
3. Move an entry from the head of the queue wherein it currently resides to the tail of another queue.
4. Provide an entry of a queue to the RTU.

5. Provide the size of a queue.
6. Update any of the fields associated to packetNumber.

Operations 1, 2, 4 and 6 above (applied to different packets at the same time) are completed in 4 cycles in a preferred embodiment of the present invention. This implies a throughput of one operation per cycle.

Some prioritization is necessary when two or more operations could start to be executed at the same time. From highest to lowest priority, these events are inserting from the PMMU, dequeuing an entry, moving an entry from one queue to another queue, sending an entry to the RTU for pre-loading, or a software operation. The prioritization among the software operations is provided by design since software operations are always executed in order.

### Early QS full detection

The PMU implements a mechanism to aid in flow control between the ASIC (see element 203 in Fig. 2) and the XCaliber processor. Part of this mechanism is to detect that the QS is becoming full and, in this case, a LessThanXpacketIdEntriesInt interrupt is generated to the SPU. The software can enable this interrupt by specifying (in a IntIfLessThanXpacketIdEntries configuration register) a number  $z$  larger than 0. An interrupt is generated when  $256 - y < z$ , being  $y$  the total number of packets currently in process in XCaliber. When  $z = 0$ , the interrupt will never occur.

## **Register Transfer Unit (RTU)**

A goal of the RTU block is to pre-load an available context with information of packets alive in XCaliber. This information is the packetPage and packetNumber of the packet and some fields of its header. The selected context is owned by the PMU at the time of the pre-loading, and released to the SPU as soon as it has been pre-loaded. Thus, the SPU does not need to perform the costly load operations to load the header information and, therefore, the overall latency of processing packets is reduced.

The RTU receives from the QS a packet identifier (packetPage, packetNumber) and the number of the queue from which the packet comes from) from the QS. This identifier is created partly by the PMMU as a result of a new packet arriving to XCaliber through the network input interface (packetPage), and partly by the QS when the packetPage and device identifier are enqueued (packetNumber).

Another function of the RTU is to execute masked load/store instructions dispatched by the SPU core since the logic to execute a masked load/store instruction is similar to the logic to perform a pre-load. Therefore, the hardware can be shared for both operations. For this reason, the RTU performs either a masked load/store or a pre-load, but not both, at a time. The masked load/store instructions arrive to the RTU through the command queue (CU) block.

## **Context States**

A context can be in one of two states: PMU-owned or SPU-owned. The ownership of a context changes when the current owner releases the

context. The PMU releases a context to the SPU in three cases. Firstly, when the RTU has finished pre-loading the information of the packet into the context. Secondly, the PMU releases a context to the SPU when the SPU requests a context to the RTU. In this case, the RTU will release a context if it has one available for releasing. Thirdly, all eight contexts are PMU-owned. Note that a context being pre-loaded is considered to be a PMU-owned context.

The SPU releases a context to the RTU when the SPU dispatches an XStream RELEASE instruction.

### Pre-loading a Context

At boot time, the PMU owns 7 out of the 8 contexts that are available in the embodiment of the invention described in the present example, and the SPU owns one context. The PMU can only pre-load information of a packet to a context that it owns. The process of pre-loading information of a packet into a context is divided into two steps. A first phase to load the address (the offset within the packet memory address space), from where the packet starts. This offset points to the first byte of the two-byte value that codes the size in bytes of the packet. In the case that the packet has been time stamped and HeaderGrowthOffset is not 0, the time stamp value is located at offset-4. The offset address is computed as  $(\text{packetPage} \ll 8) | (\text{HeaderGrowthOffset} \ll 4)$ . This offset is loaded into register number StartLoadingRegister in the selected context. StartLoadingRegister is a boot-time configuration register. The packetNumber value is loaded in register number StartLoadingRegister+1.

The second phase is to load the packet header. The packet header is loaded using registers StartLoadingRegister+2, StartLoadingRegister+3, ... (as many as needed, and as long as there exist GPR registers). The PatternMatchingTable[ $q$ ] ( $q$  being the queue number associated to the packet) mask specifies how the header of the packet will be loaded into the GPR registers of the context. The PatternMatchingTable is an on-the-fly configuration register that contains masks. To obtain the header data, the RTU requests the SIU to read as many 16-byte lines of packet data as needed into the packet memory. The RTU, upon receiving the 16-byte lines from packet memory (either local or external), selects the desired bytes to load into the context using pattern mask to control this operation.

The step described immediately above of loading the packet header may be disabled by software on a per-queue basis through the on-the-fly PreloadMaskNumber configuration register. This register specifies, for each of the 32 possible queues in the QS, which mask (from 0 to 23) in the PatternMatchingTable is going to be used for the pre-loading. If a value between 24 and 31 is specified in the configuration register, it is interpreted by the RTU as not to perform.

The RTU only loads the GPR registers of a context. The required CP0 registers are initialized by the SPU. Since the context loaded is a PMU-owned context, the RTU has all the available write ports to that context (4 in this embodiment) to perform the loading.

Whenever the pre-loading operation starts, the RTU notifies this event to the SPU through a dedicated interface. Similarly, when the pre-loading operation is completed, the RTU also notified the SPU. Thus the SPU expects two notifications (start and end) for each packet pre-load. A special notification is provided to the SPU when the RTU starts and ends a pre-load in the same cycle (which occurs when the step of loading packet header is

disabled). In all three cases, the RTU provides the context number and the contents of the CodeEntryPoint configuration register associated to the packet. In the case that the PMU releases a context to the SPU because all eight contexts are PMU-owned, the contents of the CodeEntryPointSpecial are provided to the SPU. The RTU has an 8-entry table (one for each context), each entry having a packet identifier ready to be pre-loaded and a valid bit that validates the entry. The RTU selects always the valid identifier of the highest entry index to do the pre-load. When a context is associated to this identifier, the corresponding valid bit is de-asserted. The RTU pre-loads one context at a time. After loading a context, the context is released to the SPU and becomes a SPU-owned context. At this point the RTU searches its table for the next packet to be pre-loaded into a context (in case there is at least one PMU-owned context).

### Pattern-Matching Table

Figure 15 illustrates a Pattern Matching Table which is an on-the-fly configuration register that contains a set of sub-masks. The RTU can use any sub-mask (from 0 to 23) within this table for a pre-loading a context. Sub-masks can also be grouped into a larger mask containing two or more submasks.

Fig. 16 illustrates the format of a mask. A mask is a variable number (1 to 8) of sub-masks of 32x2 bits each, as shown. Every sub-mask has an associated bit (EndOfMask) that indicates whether the composite mask finishes with the corresponding sub-mask, or it continues with the next sub-mask. The maximum total number of sub-masks is 32, out of which 24 (sub-mask indexes 0 to 23) are global, which means any stream in the SPU can use and update them, and 8 are per-stream sub-masks. The per-stream sub-

masks do not have an EndOfMask bit, which is because no grouping of per-stream sub-masks is allowed.

The two 32-bit vectors in each sub-mask are named SelectVector and RegisterVector. The SelectVector indicates which bytes from the header of the packet will be stored into the context. The RegisterVector indicates when to switch to the next consecutive register within the context to keep storing the selected bytes by the SelectVector. The bytes are always right aligned in the register.

Fig. 17 shows an example of a pre-load operation using the mask in Fig. 16. A bit asserted in the SelectVector indicates that the corresponding byte of the header are stored into a register. In the example, bytes 0, 1 and 7 of the header are loaded into GPR number StartLoadingRegister +2 in bytes 0, 1 and 2, respectively (i.e. the header bytes are right-aligned when loaded into the register). A bit asserted in the RegisterVector indicates that no more header bytes are loaded into the current GPR register, and that the next header bytes, if any, are loaded into the next (consecutively) GPR register. In the example, bytes 12 and 13 of the header are loaded into GPR number StartLoadingRegister+3.

## Selecting a PMU-owned Context

There are a total of eight functional units in the PMU core. However, due to complexity-performance tradeoffs, a stream (context) can only issue instructions to a fixed set of 4 functional units.

The RTU may own at any given time several contexts. Therefore, logic is provided to select one of the contexts when a pre-load is performed, or when a context has to be provided to the SPU. This logic is defined based on how the different streams (contexts) in the PMU core can

potentially dispatch instructions to the different functional units, and the goal of the logic is to balance operations that the functional units in the SPU can potentially receive.

The selection logic takes as inputs eight bits, one per context, that indicates whether that context is PMU or SPU-owned. The logic outputs which PMU-owned context(s) that can be selected.

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,3,20,5,6,7,24,9,10,11,12,  
13,14,15,32,33,34,3,36,5,6,7,40,9,10,11,12,13,14,15,48,48,48,51,48,53,  
54,7,48,57,58,11,60,13,14,15,64,65,66,3,68,5,6,7,72,9,10,11,12,13,14,  
15,80,80,80,83,80,85,86,7,80,89,90,11,92,13,14,15,96,96,96,99,96,101,  
102,7,96,105,106,11,108,13,14,15,112,112,112,112,112,112,119,112,  
112,112,123,112,125,126,15,128,129,130,3,132,5,6,7,136,9,10,11,12,13,  
14,15,144,144,144,147,144,149,150,7,144,153,154,11,156,13,14,15,160,  
160,160,163,160,165,166,7,160,169,170,11,172,13,14,15,176,176,176,176,  
176,176,176,183,176,176,176,187,176,189,190,15,192,192,192,195,192,197,198,7,192,2  
01,202,11,204,13,14,15,208,208,208,208,208,208,208,215,208,  
208,208,219,208,221,222,15,224,224,224,224,224,224,224,231,224,224,224,235,224,237  
,238,15,240,240,240,240,240,240,240,240,240,240,240,240,240,240

The selection logic is specified with the previous list of 254 numbers. Each number is associated to a possible combination of SPU/PMU-owned context. For example, the first number corresponds to the combination '00000001', i.e. context number 0 is PMU owned and context numbers 1 to 7 are SPU owned (LSB digit corresponds to context 0, MSB digit to context 7; digit value of 0 means SPU owned, digit value of 1 means PMU owned). The second number corresponds to combination '00000010', the third to combination '00000011', and so forth up to combination '11111110'. The 19<sup>th</sup> combination ('00010011') has associated number 3 (or '00000011') in the previous list, which means that context 0 and 1 can be selected. Context 4 could also be selected, however it is not the best choice to balance the use of the functional units in the SPU core.



### Interrupt when no context is available

The RTU has a table of 8 entries named NewPacketIdTable). Entry  $p$  in this table contains a packet identifier (packetPage, packetNumber and queue number) with an RTU-priority of  $p$ , and a Valid bit that validates the identifier. When the RTU is not busy pre-loading or executing a masked load/store, it will obtain from this table the valid identifier with the highest RTU-priority. In case it exists and there is at least one PMU-owned context, the RTU will start the pre-loading of a PMU-owned context, and it will reset the Valid bit in the table.

In case there is no PMU-owned context, the RTU sits idle (assuming no software operation is pending) until a context is released by the SPU. At that point in time the RTU obtains, again, the highest valid RTU-priority identifier from the NewPacketIdTable (since a new identifier with higher RTU priority could have been sent by the QS while the RTU was waiting for a context to be released by the SPU). The Valid bit is reset and the packet information starts being pre-loaded into the available context. At this point the RTU is able to accept a packet with RTU priority  $p$  from the QS.

When an identifier with a RTU priority of  $p$  is sent by the QS to the RTU, it is loaded in entry  $p$  in the NewPacketIdTable, and the Valid bit is set. At this time, if the number of valid identifiers (without counting the incoming one) in the NewPacketIdTable is equal or larger than the current available PMU-owned contexts (without counting the context that the RTU currently might be loading), then a PacketAvailableButNoContextPriority $P$  Int interrupt is generated to the SPU.  $P$  ranges from 0 to 7, and its value is determined by a boot-time configuration flag PacketAvailableButNoContextIntMapping. If this flag is '0',  $P$  is determined by the 3-bit boot-time configuration register DefaultPacketPriority. If this flag is '1',  $P$  is the RTU

priority. However, the `PacketAvailableButNoContextPriorityPint` will not be generated if the corresponding configuration flag `PacketAvailableButNoContextPriorityPintEnable` is de-asserted.

The SPU, upon receiving the interrupt, decides whether or not to release a context that it owns so that the RTU can pre-load the `packetPage`, `packetNumber` and header information of the new packet.

When the RTU generates a `PacketAvailableButNoContextPriorityPint` interrupt, it may receive after a few cycles a context that has been released by the SPU. This context, however, could have been released when, for example, one of the streams finished the processing of a packet. This can happen before the interrupt service routine for the `PacketAvailableButNoContextPriorityPint` interrupt finishes. Thus, when a context is released due to the ISR completion, the packet pre-load that originated the interrupt already might have used the context first released by another stream in the SPU. Thus, the context released due to the interrupt will be used for another (maybe future) packet pre-load. If no other entry is valid in the `NewPacketIdTable`, the context is be used and sits still until either an identifier arrives to the RTU or the SPU requesting a context to the RTU.

Whenever a context becomes SPU-owned, and the RTU has a pre-load pending, the RTU selects the most priority pending pre-load (which corresponds to the highest-valid entry in the `NewPacketTable`), and will start the preload. If the `PacketAvailableButNoContextPriorityint` interrupt associated to this level was asserted, it gets de-asserted when the pre-load starts.

## Software Operations on the RTU

Software executes a number of instructions that affect the RTU.

Following is a list of all operations that can be generated to the RTU as a result of dispatch by the SPU core of an XStream packet instruction. The operations arrive to the RTU through the command queue (CU), along with the context number associated to the stream that issued the instruction:

1. Release( $c$ ): context number  $c$  becomes PMU owned.
2. GetContext: the RTU returns the number of a PMU-owned context number. This context, if it exists, becomes SPU owned and a success flag is returned asserted; otherwise it is return de-asserted, in which case the context number is meaningless.
3. MaskedLoad( $r, a, m$ ), MaskedStore( $r, a, m$ ): the SPU core uses the RTU as a special functional unit to execute the masked load/store instructions since the logic to execute a masked load/store instruction is similar to the logic to perform a pre-load. Therefore, the hardware can be shared for both operations. For this reason, the RTU performs either a masked load/store or a pre-load, but not both at a time. For either the masked load or masked store, the RTU will receive the following parameters:
  - (a) A mask number  $m$  that corresponds to the index of the first submask in the PatternMatchingTable to be used by the masked load/store operation.
  - (b) A 36-bit address  $a$  that points to the first byte in (any) memory to which the mask will start to be applied.

(c) A register number  $r$  (within the context number provided) that corresponds to the first register involved in the masked load/store operation. Subsequent registers within the same context number will be used according to the selected mask.

5 For masked load/store operations, the mask can start to be applied at any byte of the memory, whereas in a pre-load operation (a masked-load like operation) the mask will always be applied starting at a 16-byte boundary address since packet data coming from the network input interface is always stored in packet memory starting at the LSB byte in a 16-byte line.

10 The MaskedLoad, MaskedStore and GetContext operations communicate to the SPU when they complete through a dedicated interface between the RTU and the SPU. The RTU gives more priority to a software operation than packet pre-loads. Pre-loads access the packet memory whereas the masked load/store may access any memory in the system as long as it is not cacheable or write-through. If not, results are undefined.

The RTU is able to execute a GetContext or Release command while executing a previous masked load/store command.

## 20 Programming Model

Software can configure, either at boot time or on the fly, several of the features of the PMU. All of the features configurable at boot time only, and some configurable on the fly, must happen only when the SPU is running in a single-stream mode. If not, results are undefined. The PMU does not check in which mode the SPU is running.

25 Software can update some of the information that the PMU maintains for a given packet, and also obtain this information. This is accomplished by

software through new XStream packet instructions that are the subject of separate patent applications. These instructions create operations of three different types (depending on which block of the PMU the operation affects, whether PMMU, QS or RTU) that will be executed by the PMU. Some of the operations require a result from the PMU to be sent back to the SPU.

The packet memory and configuration space are memory mapped. The SIU maintains a configuration register (16MB aligned) with the base address of the packet memory, and a second configuration register with the base address of EPM. Software sees the packet memory as a contiguous space. The system, however, allows the EPM portion of the packet memory to be mapped in a different space.

The SIU also maintains a third configuration register with the base of the PMU configuration register space. All the load/store accesses to LPM and configuration space performed by the SPU reach the PMU through the SIU. The SIU determines to which space the access belongs, and lets the PMU know whether the access is to LPM or to the PMU configuration space. Accesses to the EPM are transparent to the PMU.

The PMU can interrupt the SPU when certain events happen. Software can disable all these interrupts through configuration registers.

## Configuration Registers

The configuration registers of the PMU reside in the PMU Configuration Space of the XCaliber address space. The base address of this space is maintained by the SIU and does not need to be visible by the PMU. The SIU notifies to the PMU with a signal when a read/write access

performed by the SPU belongs to this space, along with the information needed to update the particular register on a write access.

Some of the PMU configuration registers can be configured only at boot time, and some can be configured on the fly. All boot-time configurable and some on-the-fly configurable registers need to be accessed in single-stream mode. A boot-time configurable register should only be updated if the PMU is in reset mode. Results are undefined otherwise. The PMU will not check whether the SPU is indeed in single-stream mode when a single-stream mode configuration register is updated. All the configuration registers come up with a default value after the reset sequence.

In the instant embodiment 4KB of the XCaliber address space is allocated for the PMU configuration space. In XCaliber's PMU, some of these configuration registers are either not used or are sparsely used (i.e. only some bits of the 32-bit configuration register word are meaningful). The non-defined bits in the PMU configuration space are reserved for future PMU generations. Software can read or write these reserved bits but their contents, although fully deterministic, are undefined.

Fig. 18 shows the PMU Configuration Space, which is logically divided into 32-bit words. Each word or set of words contains a configuration register.

Figs. 19a and 19b are two parts of a table showing mapping of the different PMU configuration registers into the different words of the configuration space. The block owner of each configuration register is also shown in the table.

Following is the list of all configuration registers in this particular embodiment along with a description and the default value (after PMU reset). For each of the configuration registers, the bit width is shown in parenthesis. Unless otherwise specified, the value of the configuration

register is right aligned into the corresponding word within the configuration space.

### Boot-time Only Configuration Registers:

5

#### 1. Log2InputQueues (5)

(a) Default Value: 0

(b) Description: Number of queues in the QS used as input queues (i.e. number of queues in which packetPages/deviceIds from the PMMU will be inserted).

10

#### 2. PriorityClustering (3)

(a) Default Value: 5 (32 clusters)

(b) Description: Specifies how the different queues in the QS are grouped in priority clusters (0: 1 cluster, 1: 2 clusters, 2: 4 clusters, ..., 5: 32 clusters).

15

#### 3. HeaderGrowthOffset (4)

(a) Default Value: 0

(b) Description: Number of empty 16-byte chunks that will be left in front of the packet when it is stored in packet memory. Maximum value is 15 16-byte chunks. Minimum is 0.

20

#### 4. TailGrowthOffset (6)

(a) Default Value: 0

(b) Description: Number of empty 16-byte chunks that will be left at the end of the packet when it is stored in packet memory. Maximum value is 63 16-byte chunks. Minimum is 0.

25

5. PacketAvailableButNoContextIntMapping (1)

(a) Default Value: 0

(b) Description: Specifies the  $P$  in the PacketAvailableButNoContextPriority $P$ Int interrupt, if enabled. The possible values are:

(1) 0:  $P$  is specified by the DefaultPacketPriority register.

(2) 1:  $P$  is the RTU priority.

6. StartLoadingRegister (5)

(a) Default Value: 1

(b) Description: Determines the first GPR register number to be loaded by the RTU when performing the background load of the packet header on the chosen context. In this register, the value  $(\text{packetPage} \ll 8) | (\text{HeaderGrowthOffset} \ll 4)$  is loaded. The packetNumber is loaded in the next GPR register. The following GPR registers will be used to pre-load the packet header data following PatternMatchingMask0 mask if this feature is enabled.

7. PreloadMaskNumber (32x5)

(a) Default Value: mask 31 for all queues (i.e. pre-load of header is disabled).

(b) Description: It specifies, for each of the 32 possible queues in the QS, which mask in the PatternMatchingTable is going to be used for pre-loading.

Figs. 19a-c show a mapping of the PreloadMaskNumber configuration register.



The configuration registers described above are the boot-time-only configuration registers in the instant example. Immediately below are listed the On-The-Fly configuration registers.

5

## Single-stream Configuration Registers

### 1. OverflowEnable (1)

(a) Default Value: 0

10

(b) Description: Enables/disables the overflow of packets in case they do not fit into LPM. When disabled, these packets are dropped.

### 2. PatternMatchingTable (24x(32x2+1))

(a) Default Value (per each of the 24 entries):

15

(1) SelectVector: select all bytes

(2) RegisterVector: store 4 consecutive bytes per register

(3) EndOfMask: 1

20

(b) Description: It specifies, for masked load/store operations, which bytes to load/store and in which (consecutive) registers. Mask 0 of this table is used by the RTU to pre-load, in the background, some bytes of the header of the packet in one of the available contexts.

There are a total of 24 masks.

(c) Note: Mask 0 needs to be written when the PMU is freezed (see Section 0), otherwise results are undefined.

25

Fig. 21 illustrates the PatternMatchingTable described immediately above.

30

### 3. Freeze (1)

(a) Default Value: 1

(b) Description: Enables/disables the freeze mode.

#### 4. Reset (1)

(a) Default Value: 0

(b) Description: When set to 1, forces the PMU to perform the reset sequence. All packet data in the PMU will be lost. After the reset sequence all the configuration registers will have the default values.

### Multi-stream Configuration Registers

#### 1. ClearErrorD ( $D = 0, 1$ )

(a) Default Value: 0

(b) Description: When written by software (with any data), the packet error condition detected on device identifier  $D$  is cleared.

#### 2. PacketAvailableButNoContextPriorityPintEnable (8) [ $P = 0..7$ ]

(a) Default Value: 0 (for all levels)

(b) Description: Enables/disables the PacketAvailableButNoContextPriorityPint interrupt.

#### 3. AutomaticPacketDropIntEnable (1)

(a) Default Value: 1

(b) Description: Enables/disables the AutomaticPacketDropInt interrupt.

#### 4. TimeStampEnable (1)

(a) Default Value: 0

(b) Description: Enables/disables the time stamp of packets. When enabled and HeaderGrowthOffset is greater than 0, a 4-byte time stamp is appended to the packet before it is written into the packet memory.

5

#### 5. PacketErrorIntEnable (1)

(a) Default Value: 0

(b) Description: Enables/disables the PacketErrorInt interrupt.

10

#### 6. VirtualPageEnable (9x4)

(a) Default Value: all virtual pages enabled for all blocks.

(b) Description: Enables/disables the virtual pages for each of the 4 blocks that the LPM is divided into. There are up to 9 virtual pages, from 256 bytes (enabled by the LSB bit) up to 64K bytes (enabled by the MSB bit), with all power-of-two sizes in between.

15

Fig. 22 illustrates the VirtualPageEnable register.

20

#### 7. OverflowAddress (24)

(a) Default Value: 0x40000 (the first atomic page in the EPM)

(b) Description: the 16 MSB bits correspond to the atomic page number in packet memory into which the packet that is overflowed will start to be stored. The 8 LSB are hardwired to '0' (i.e. any value set by software to these bits will be disregarded). OverflowAddress is then the offset address within the 16MB packet memory. The SIU will translate this offset into the corresponding physical address into the EPM. The first 1K atomic pages of the packet memory

25

correspond to the LPM. If software sets the 16 MSB of OverflowAddress to 0..1023, results are undefined. When a packet is overflowed, the 16 MSB bits of OverflowAddress become the packetPage for that packet. The SPU allows the next packet overflow when it writes into this configuration register.

#### 8. IntIfNoMoreXsizePages (4)

- (a) Default Value: 0xF (i.e. the interrupt will never be generated)
- (b) Description: Specifies the index of a virtual page (0:256 bytes, 1:512 bytes, ... , 8:64K bytes, 9-15: no virtual page). Whenever the PMMU detects that there are no more virtual pages of that size in all the LPM, the NoMoreThanXSizePagesInt interrupt will be generated to the SPU.

#### 9. IntIfLessThanXpacketIdEntries (9)

- (a) Default Value: 0
- (b) Description: Minimum number of entries in the QS available for new packet identifiers. If the actual number of available entries is less than this number, an interrupt will be generated to the SPU. If this number is 0, the LessThanXPacketIdEntriesInt interrupt will not be generated.

#### 10. DefaultPacketPriority (3)

- (a) Default Value: 0
- (b) Description: Provides the priority level for the PacketAvailableButNoContextInt interrupt when PacketAvailableButNoContextMapping is 0.

11. ContextSpecificPatternMatchingMask: (8x(32x2))

(a) Default Value:

(1) SelectVector: select all bytes

(2) RegisterVector: store 4 bytes in each register

(EndOfMask is hardwired to 1)

(b) Description: It specifies, for masked load/store operations, which bytes to load/store and in which (consecutive) registers. Software will guarantee that a stream only access its corresponding context-specific mask.

Fig. 23 illustrates the ContextSpecificPatternMatching mask configuration register.

12. FirstInputQueue (5)

(a) Default Value: 0

(b) Description: Specifies the smallest number of the queue into which packets from the PMMU will be inserted.

13. SoftwareOwned (4)

(a) Default Value: 0 (not software owned)

(b) Description: one bit per LPM block. If '1', the block is software owned, which implies that the memory of the block is managed by software, and that the VirtualPageEnable bits for that block are a don't care.

14. MaxActivePackets (32x9)

(a) Default Value: 256 for each of the queues.

(b) Description: Specifies, for each queue  $q$ , a value between 0 and 256 that corresponds to the maximum number of packets within queue  $q$  that can be being processed by the SPU.

Fig. 24 illustrates the MaxActivePackets configuration register.

5

#### 15. CodeEntryPoint (32x30)

(a) Default Value: 0 for each of the queues.

(b) Description: The contents of the CodeEntryPoint register associated to queue  $q$  are sent to the SPU when a context is activated which has been pre-loaded with a packet that resides in queue  $q$ .

10

#### 16. CodeEntryPointSpecial (30)

(a) Default Value: 0

(b) Description: The contents of this register are sent to the SPU when a context is activated due to the fact that all the contexts become PMU-owned.

15

#### 17. Bypass Hooks (9)

(a) Default Value: 0

(b) Description: See Fig. 32. Each bit activates one hardware bypass hook. The bypass hook is applied for as many cycles as the corresponding bit in this register is asserted.

20

#### 18. InternalStateWrite (12)

(a) Default Value: 0

(b) Description: See Fig. 33. Specifies one word of internal PMU state. The word of internal state will be available to software when

25

reading the InternalStateRead configuration register. The InternalStateWrite configuration register is only used in one embodiment to debug the PMU.

5

## Read-only Registers

### 1. SizeOfOverflowedPacket (16)

(a) Default Value: 0

(b) Description: Whenever the PMU has to overflow a packet, this register will contain the size in bytes of that packet.

10

### 2. TimeCounter (64)

(a) Default Value: 0

(b) Description: Contains the number of core clock cycles since the last reset of the PMU.

15

The TimeCounter configuration register is illustrated in Fig. 25.

### 3. StatusRegister (8)

(a) Default Value: 1

(b) Description: Contains the state of the PMU. This register is polled by the SPU to figure out when the reset or freeze has completed (Freeze and Reset bits), or to figure out the source of packet error per inbound device identifier (Err: 1 - error, 0 - no error; EPM: 1 - error has occurred while packet is overflowed to EPM, 0 - error has occurred while packet is being stored in LPM; PSM: 1 - error due to a packet size mismatch, 0 - error due to a bus error).

20

25

Fig. 26 illustrates the StatusRegister configuration register

## Interrupts

The PMU can interrupt the SPU when certain events happen. Software can disable all these interrupts using some of the configuration registers listed above. Moreover, each stream can individually mask these interrupts, which is the subject of a separate patent application. The list of interrupts that the PMU generate are as follows:

### 1. OverflowStartedInt

(a) Interrupt Condition: When the PMMU cannot store the incoming packet into the LocalPacketMemory, it will overflow the packet to the ExternalPacketMemory through the SIU.

(b) Disable Condition: OverflowEnable = '0'

### 2. NoMorePagesOfXSizeInt

(a) Interrupt Condition: When no more free virtual pages of the size indicated in IntIfNoMoreXSizePages are available.

(b) Disable Condition: IntIfNoMoreXSizePages = {10,11,12,13,14,15}.

### 3. LessThanXPacketIdEntriesInt

(a) Interrupt Condition: When the actual number of available entries in the QS is less than IntIfLessThanXPacketIdEntries.

(b) Disable Condition: IntIfLessThanXPacketIdEntries = 0

### 4. PacketAvailableButNoContextPriorityPint ( $P=0..7$ )



(a) Interrupt Condition: When a packet identifier is received by the RTU from the QS but there is no available context.

(b) Disable Condition:

PacketAvailableButNoContextPriorityPIntEnable = '0'

5

#### 5. AutomaticPacketDropInt

(a) Interrupt Condition: When a packet cannot be stored in LPM and OverflowEnable = '0'.

(b) Disable Condition: AutomaticPacketDropIntEnable = '0'

10

#### 6. PacketErrorInt

(a) Interrupt Condition: When the actual size of the packet received from the ASIC does not match the value in the first two bytes of the ASIC-specific header, or when a bus error has occurred.

(b) Disable Condition: PacketErrorIntEnable = '0'

15

20

Interrupts to the SPU in this embodiment are edge-triggered, which means that the condition that caused the interrupt is cleared in hardware when the interrupt is serviced. This also implies that the condition that causes the interrupt may happen several times before the interrupt is served by the SPU. Therefore, the corresponding interrupt service routine will be executed only once, even though the condition that causes the interrupt has happened more than once.

25

This behavior is not desirable for some of the interrupts. For these cases, a special interlock mechanism is implemented in hardware that

guarantees that the condition will not happen again until the interrupt has been serviced.

An example of the special interlock mechanism is the case of the OverflowStartedInt and PacketAvailableButNoContextPriorityPInt interrupts. In the first case, when a packet is overflowed, no other packet are overflowed until the software writes a new address in the on-the-fly configuration register OverflowAddress. If a packet has been overflowed but the OverflowAddress register still has not been written by the software, any subsequent packet that would have otherwise been overflowed because it does not fit in the LPM must be dropped.

For the 8 PacketAvailableButNoContextPriorityPInt ( $P = 0..7$ ) interrupts, the PMU architecture implicitly guarantees that no multiple conditions (per each  $P$ ) will occur. This is guaranteed by design since:

- (a) the PacketAvailableButNoContextPriorityPInt interrupt is only generated when a packet identifier of RTU priority  $P$  arrives to the RTU, and
- (b) at most, only one packet identifier with RTU priority  $P$  resides in the RTU.

The other interrupts can suffer from the multiple condition effect. Therefore, software should not rely on counting the number of times a given type of interrupt happens to figure out exactly how many times that condition has occurred.

## Protection Issues

The architecture of the PMU in the instant embodiment creates the following protection issues:

1. An stream could read/write data from a packet other than the one it is processing. An stream has access to all the packet memory, and there is no mechanism to prevent an stream from accessing data from a totally unrelated packet unless the packet memory is mapped as kernel space.
2. Since the configuration registers are memory mapped, any stream could update a configuration register, no matter whether the SPU is in single-stream mode or not. In particular, any stream could freeze and reset the PMU.
3. Whenever a packet is completed or moved with reactivation, nothing prevents software from continuing “processing” the packet.

### **Command Unit (CU)**

Software can update some information that the PMU maintains for a given packet and obtain this information. This is accomplished by software through some of the new XStream packet instructions referred to above. Some of these instructions are load-like in the sense that a response is required from the PMU. Others are store-like instructions, and no response is required from the PMU.

Fig. 27 is a diagram of Command Unit 213 of Fig. 2, in relation to other blocks of the XCaliber processor in this example, all of which bear the same element numbers in Fig. 27 as in Fig. 2. The SPU dispatches, at most, two packet instructions per cycle across all contexts (one instruction per cluster of the SPU). The type of the packet instruction corresponds to the

PMU block to which the instruction affects (PMMU, QS or RTU). When the SPU dispatches a packet instruction, a single command to the PMU is generated and inserted into one of three different queues in the CU block (one queue per PMU block to which the command goes). Commands to the PMU are issued to PMMU command queue 2703, those to the QS go to QS command queue 2705, and command to the RTU go to the RTU command queue 2707. Each queue can hold up to 8 commands. The SPU only dispatches a command to the CU if there are enough free entries in the corresponding queue.

The CU is responsible for dispatching the commands to the respective blocks, and gathering the responses (if any) in an 8-entry ResponseQueue 2709, which queues responses to be returned to the SPU. The CU can receive up to three responses in a given cycle (one from each of the three blocks). Since (a) only one outstanding packet instruction is allowed per stream, (b) the Response Queue has as many entries as streams, (c) only one command to the PMU is generated per packet instruction, and (d) only one response is generated per each load-like command, it is guaranteed that there will be enough space in the ResponseQueue to enqueue the responses generated by the PMU blocks. The ResponseQueue should be able to enqueue up to two commands at a time.

CU 213 also receives requests from SIU 107 to update the configuration registers. These commands are also sent to the PMMU, RTU and QS blocks as commands. The PMMU, QS, and RTU keep a local copy of the configuration registers that apply to them. The CU keeps a copy as well of all the configuration registers, and this copy is used to satisfy the configuration register reads from the SIU.

For read-only configuration registers, a special interface is provided between the CU and the particular unit that owns the read-only configuration

register. In XCaliber's PMU, there exists two read-only configuration registers: one in the PMMU block (SizeOfOverflowedPacket) and the other one in the CU block (StatusRegister). Whenever the PMMU writes into the SizeOfOverflowedPacket register, it notifies the CU and the CU updates its local copy.

Commands in different queues are independent and can be executed out of order by the PMU. Within a queue, however, commands are executed in order, and one at a time. The PMU can initiate the execution of up to 3 commands per cycle. The PMMU and QS blocks give more priority to other events (like the creation of a new packetPage when a new packet arrives –PMMU-, or the extraction of a packet identifier because it needs to be sent out –QS-) than to the commands from the SPU. This means that a command that requests some data to be sent back to the SPU may take several cycles to execute because either the PMMU or QS might be busy executing other operations.

RTU 227 has two sources of commands: from the QS (to pre-load packet information into an available context) and from the SPU (software command). The RTU always gives more priority to SPU commands. However, the RTU finishes the on-going context pre-load operation before executing the pending SPU command.

### Command/response formats

A command received by the CMU has three fields in the current embodiment:

1. Context number, which is the context associated to the stream that generated the command.

2. Command opcode, which is a number that specifies the type of command to be executed by the PMU.
3. Command data, which is the different information needed by the PMU to execute the command specified in the command opcode field.

5

The PMU, upon receiving a command, determines to which of the command queues the command needs to be inserted. A command inserted in any of the queues has a similar structure as the command received, but the bit width of the opcode and the data will vary depending on the queue. The table of Fig. 28 shows the format of the command inserted in each of the queues. Not included are the Read Configuration Register and Write Configuration Register commands that the CU sends to the PMMU, QS and RTU blocks.

10

11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967  
1968  
1969  
1970  
1971  
1972  
1973  
1974  
1975  
1976  
1977  
1978  
1979  
1980  
1981  
1982  
1983  
1984  
1985  
1986  
1987  
1988  
1989  
1990  
1991  
1992  
1993  
1994  
1995  
1996  
1997  
1998  
1999  
2000  
2001  
2002  
2003  
2004  
2005  
2006  
2007  
2008  
2009  
2010  
2011  
2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025  
2026  
2027  
2028  
2029  
2030  
2031  
2032  
2033  
2034  
2035  
2036  
2037  
2038  
2039  
2040  
2041  
2042  
2043  
2044  
2045  
2046  
2047  
2048  
2049  
2050  
2051  
2052  
2053  
2054  
2055  
2056  
2057  
2058  
2059  
2060  
2061  
2062  
2063  
2064  
2065  
2066  
2067  
2068  
2069  
2070  
2071  
2072  
2073  
2074  
2075  
2076  
2077  
2078  
2079  
2080  
2081  
2082  
2083  
2084  
2085  
2086  
2087  
2088  
2089  
2090  
2091  
2092  
2093  
2094  
2095  
2096  
2097  
2098  
2099  
2100  
2101  
2102  
2103  
2104  
2105  
2106  
2107  
2108  
2109  
2110  
2111  
2112  
2113  
2114  
2115  
2116  
2117  
2118  
2119  
2120  
2121  
2122  
2123  
2124  
2125  
2126  
2127  
2128  
2129  
2130  
2131  
2132  
2133  
2134  
2135  
2136  
2137  
2138  
2139  
2140  
2141  
2142  
2143  
2144  
2145  
2146  
2147  
2148  
2149  
2150  
2151  
2152  
2153  
2154  
2155  
2156  
2157  
2158  
2159  
2160  
2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2

The RTU notifies the SPU, through a dedicated interface that bypasses the CU (path 2711 in Fig. 27), of the following events:

1. A masked load/store operation has finished. The interface provides the context number.
2. A GetContext has completed. The context number associated to the stream that dispatched the GetContext operation, and the context number selected by the RTU is provided by the interface. A success bit is asserted when the GetContext succeeded; otherwise it is de-asserted.
3. A pre-load either starts or ends. The context number and the priority associated to the packet is provided to the SPU.

### **Reset and freeze modes**

The PMU can enter the reset mode in two cases:

1. SPU sets the Reset configuration flag.
2. XCaliber is booted.

The PMU can also enter the freeze mode in two cases:

1. SPU sets the Freeze configuration flag.
2. PMU finishes the reset sequence.

The reset sequence of the PMU takes several cycles. During this sequence, the Reset bit in the StatusRegister configuration register is set. After the reset sequence, all the configuration registers are set to their default values, and the PMU enters the freeze mode (the Reset bit in the StatusRegister is reset and the Freeze bit is set). When this is done, the SPU

resets the Freeze configuration flag and, from that time on, the PMU runs in the normal mode.

When the SPU sets the Freeze configuration flag, the PMU terminates the current transaction or transactions before setting the Freeze bit in the StatusRegister. Once in the freeze mode, the PMU will not accept any data from the network input interface, send any data out through the network output interface, or pre-load any packet

The PMU continues executing all the SPU commands while in freeze mode.

The SPU needs to poll the StatusRegister configuration register to determine in which mode the PMU happened to be (reset or freeze) and to detect when the PMU changes modes.

The CU block instructs the rest of the blocks to perform the reset and the freeze. The following is the protocol between the CU and any other block when the CU receives a write into the reset and/or freeze configuration bit:

1. The CU notifies to some of the blocks that either a freeze or a reset needs to be performed.
2. Every block performs the freeze or the reset. After completion, the block signals back to the CU that it has completed the freeze or reset.
3. The CU updates the StatusRegister bits as soon as the reset or freeze has been completed. Software polls the StatusRegister to determine when the PMU has completely frozen.

The different blocks in the PMU end the freeze when:

1. IB, LPM, CU and QS do not need to freeze.
2. As soon as the PMMU finishes uploading inbound packets, if any, and downloading outbound packets, if any.



3. As soon as the RTU has finished the current pre-load operation, if any.
4. As soon as the OB is empty.

While in freeze mode, the blocks will not:

1. start uploading a new packet; start downloading a completed packet; or generate interrupts to the SPU (PMMU)
2. pre-load a context or generate interrupts to the SPU (RTU).

If software writes a '1' in the Freeze/Reset configuration register and then writes a '0' before the PMU froze or reset, results are undefined. Once the PMU starts the freeze/reset sequence, it completes it.

### **Performance Counters Interface**

The PMU probes some events in the different units. These probes are sent to the SIU and used by software as performance probes. The SIU has a set of counters used to count some of the events that the PMU sends to the SIU. Software decides which events throughout the XCaliber chip it wants to monitor. Refer to the SIU Architecture Spec document for more information on how software can configure the performance counters.

Fig. 30 shows a performance counter interface between the PMU and the SIU. Up to 64 events can be probed within the PMU. All 64 events are sent every cycle to the SIU (*EventVector*) through a 64-bit bus.

Each of the 64 events may have associated a value (0 to 64K-1). Software selects two of the events (*EventA* and *EventB*). For each of these two, the PMU provides the associated 16-bit value (*EventDataA* and

*EventDataB*, respectively) at the same time the event is provided in the *EventVector* bus.

Events are level-triggered. Therefore, if the PMU asserts the event for two consecutive cycles, the event will be counted twice. The corresponding signal in the *EventVector* will be asserted only if the event occurs, and for as many cycles as the event condition holds.

The SIU selects which events are actually counted (based on how software has programmed the SIU). If the SIU decides to count an event number different from *EventA* or *EventB*, a counter within the SIU counts the event for as many cycles the corresponding bit in the *EventVector* is asserted. If the events monitored are *EventA* and/or *EventB*, the SIU, in addition to counting the event/s, increments another counter by *EventDataA* and/or *EventDataB* every time the event occurs.

Fig. 31 shows a possible implementation of the internal interfaces among the different blocks in PMU 103. CU acts as the interface between the PMU and SIU for the performance counters. CU 213 distributes the information in *EventA* and *EventB* to the different units and gathers the individual *EventVector*, *EventDataA* and *EventDataB* of each of the units.

The CU block collects all the events from the different blocks and send them to the SIU. The CU interfaces to the different blocks to notify which of the events within each block need to provide the *EventDataA* and/or *EventDataB* values.

Performance events are not time critical, i.e. they do not need to be reported to the SIU in the same cycle they occur.

Figs. 34 through 39 comprise a table that lists all events related to performance counters. These events are grouped by block in the PMU. The event number is shown in the second column. This number corresponds to

the bit in the *EventVector* that is asserted when the event occurs. The third column is the event name. The fourth column shows the data value associated to the event and its bit width in parentheses. The last column provides a description of the event.

5           The CU block collects all of the events from the different blocks and sends them to the SIU. The CU interfaces to the different blocks to notify which of the events within each block need to provide the *EventDataA* and the *EventDataB* values.

10           Performance events are not time critical, i.e. they do not need to be reported to the SIU in the same cycle that they occur.

## Debug Bypasses and Trigger Events

15           Hardware debug hooks are implemented in the PMU to help debugging of the silicon. The debug hooks are divided into two categories:

20           1. Bypass hooks: will bypass potentially faulty functions. Instead of the faulty results generated by these functions (or, in some cases, no result at all), the bypass hook will provide at least some functionality that will allow other neighboring blocks to be tested.

          2. Trigger events: when a particular condition occurs in the PMU (trigger event), the PMU will enter automatically in single-step mode until, through the OCI Interface (Section ), the SIU sends a command to the PMU to exit the single-step mode.

25           Moreover, the PMU has the capability of being single-stepped. A signal (*SingleStep*) will come from the OCI Interface. On a cycle-by-cycle basis, the different blocks of the PMU will monitor this signal. When this signal is de-asserted, the PMU will function normally. When *SingleStep* is asserted, the PMU will not perform any work: any operation on progress will be held until

the signal is de-asserted. In other words, the PMU will not do anything when the signal is asserted. The only exception to this is when a block can lose data (an example could be in the interface between two block: a block A sends data to a block B and assumes that block B will get the data in the next cycle; if *SingleStep* is asserted in this cycle, block B has to guarantee that the data from A is not lost).

### Bypass hooks

The different bypass hooks in the PMU are activated through the on-the-fly BypassHooks configuration register. Fig. 40 is a table illustrating the different bypass hooks implemented in the PMU. The number of each hook corresponds to the bit number in the BypassHooks register. The bypass hook is applied for as many cycles as the corresponding bit in this register is asserted.

### Trigger Events

The following is a list of trigger events implemented in the PMU.

1. A new packet of size  $s$  bytes is at the head of the IBU.
  - (a)  $s = 0$ : any packet.
2. A packetId from source  $s$  with packetPage  $pp$  is inserted in queue  $q$  in the QS.
  - (a)  $s = 0$ : PMM,  $s = 1$ : QS,  $s = 2$ : CMU;  $s = 3$ : any
  - (b)  $pp = 0x10000$ : any
  - (c)  $q = 33$ : any
3. A packetId from queue  $q$  with packetPage  $pp$  and packetNumbet  $pn$  is sent to RTU.

(a)  $pp = 0x10000$ : any

(b)  $q = 33$ : any

(c)  $pn = 256$ : any

4. A packetId with packetPage  $pp$  and packetNumber  $pn$  reaches the head of queue  $q$  in the QS.

(a)  $pp = 0x10000$ : any

(b)  $q = 33$ : any

(c)  $pn = 256$ : any

5. A packet with RTU priority  $p$  and packetPage  $pp$  and packetNumber  $pn$  is pre-loaded in context  $c$ .

(a)  $pp = 0x10000$ : any

(b)  $q = 33$ : any

(c)  $pn = 256$ : any

(d)  $c = 8$ : any

6. A packetId from queue  $q$  with packetPage  $pp$  and packetNumber  $pn$  is sent for downloading to PMM.

(a)  $pp = 0x10000$ : any

(b)  $q = 33$ : any

(c)  $pn = 256$ : any

7. A packetId with packetPage  $pp$  and packetNumber  $pn$  reaches the head of queue  $q$  in the QS.

(a)  $pp = 0x10000$ : any

(b)  $q = 33$ : any

(c)  $pn = 256$ : any

8. Packet command  $pc$  is executed by block  $b$ .

(a)  $pc = 0$ : GetSpace;  $pc = 1$ : FreeSpace;  $pc = 2$ : InsertPacket;  $pc = 3$ :

ProbePacket;  $pc = 4$ : ExtractPacket;  $pc = 5$ : CompletePacket;  $pc = 6$ :

UpdatePacket;  $pc = 7$ : MovePacket;  $pc = 8$ : ProbeQueue;  $pc = 9$ :

GetContext;  $pc = 10$ : ReleaseContext;  $pc = 11$ : MaskedLoad;  $pc = 12$ :

MaskedStore;  $pc = 13$ : any

(b)  $b = 0$ : RTU;  $b = 1$ : PMM;  $b = 2$ : QS;  $b = 3$ : any

### **Detailed Interfaces with the SPU and SIU**

5

The architecture explained in the previous sections is implemented in the hardware blocks shown in Fig. 41:

10

#### **SPU-PMU Interface**

Figs. 42 - 45 describe the SPU-PMU Interface.

FIG. 41

#### **SPU-PMU Interface**

Figs. 46-49 describe the SIU-PMU Interface.

20

The specification above describes in enabling detail a Packet Memory Unit (PMU) for a Multi-Streaming processor adapted for packet handling and processing. Details of architecture, hardware, software, and operation are provided in exemplary embodiments. It will be apparent to the skilled artisan that the embodiments described may vary considerably in detail without departing from the spirit and scope of the invention. It is well-known, for example, that IC hardware, firmware and software may be accomplished in a variety of ways while still adhering to the novel architecture and functionality taught.

25

30

### **Insulative Packet Storage**

In one aspect of the present invention, the inventor provides a method for insuring that consecutively stored data packets are positioned in memory in such a way that they may be allowed to grow in size to a certain amount during processing without requiring packet move operations.

As previously described the hardware-controlled Local Packet Memory is partitioned into sectors of 256 bytes. Therefore, under normal circumstances, if a packet of 257 bytes arrives, that packet will be assigned to two separate available 256 byte regions, one of which will be occupied by a single byte. If the software is aware of the 256 byte partition, then there is a built-in tail growth offset for this packet, and if the packet grows the new data can be stored in the second of the two assigned regions. It is, however, more desirable that the software not be aware of the partition size of the hardware-controlled memory, because, in new and later implementations of the system the partition size might change, then the software would not work.

A software-configurable header growth and tail growth offset is then desirable because it hides the hardware implementation from the software, and allows software to be safely ported to a next generation. Such a configurable header and tail growth offset system is described below.

Referring now to Fig. 8 of S/N 09/737,375, a portion of packet memory is illustrated containing a data packet for processing. It is described above with regard to Fig. 8 that a data packet can grow or shrink as a result of SPU processing. Moreover, a data packet may also retain the same size in bytes of data. Fig. 8 illustrates both head growth and shrink as well as tail growth and shrink possibilities. There may also be a case where the header of a data packet grows during processing while the tail portion shrinks, and

so on. It is described in 09/737,375 that the PMU can be configured to allocate extra memory space acting as insulation against a packet overwriting a next packet stored in the memory. It has also been described that virtual and atomic pages are allocated by the PMMU for the purpose of noting where in memory packets will be stored. An offset from a beginning address denotes the beginning of the data packet and the size of the data packet determines how many pages of memory it will occupy. Therefore the PMMU can select a space in memory beginning with an offset that is large enough to hold the data given the known size of a data packet.

Fig. 50 is a block diagram illustrating a portion of memory space 5000 for holding data packets according to prior art. Packet memory 5000 is illustrated herein and is analogous to the packet memory described with respect to Fig. 8 above. This example is intended to illustrate possible overwriting of data resulting in corrupt data packets. Packet memory 5000 can be assumed in this example to be local packet memory (LPM) described in S/N 09/737,375 as hardware-controlled memory. Memory 5000 has three exemplary data packets stored therein. These are data packets 5001 (Packet B), 5002 (Packet A), and 5003 (packet C). Data packets 5001-5003 will hereinafter be referred to by their alphabetical labels A-C.

Data packets B and C are illustrated as crosshatched diagonally in the same direction. Data packet A is illustrated as crosshatched diagonally in the reverse direction from the crosshatching of packets B and C. In this example, it is packet A that is currently being processed. During processing, packet A has a Packet A head growth denoted by X and a Packet A tail growth denoted herein by Y. It can be seen in this example that the growth of packet A has forced an overwrite of data into the portion of memory 5000 occupied by packet B and into the portion of memory 5000 occupied by packet C. Had packet A stayed the same size or shrunk during processing,



there would be no conflict. However, the illustrated overwrites denoted by X and Y cause corruption of both packets B and C.

In actual practice, the overwrite would not take place, because software running in the SPU would have moved packet A into another location of memory large enough to accept the packet growth that occurred during processing, before actually overwriting data into allotted memory space. It is described above with reference to Fig. 8 that a packet move operation contains the following steps:

1. The software running on the SPU requests from the PMU for a chunk of contiguous space of the new packet size. The PMU will return a new packetPage that identifies (points to) this new space.
2. The SPU software writes the data into the new memory space.
3. The SPU software renames the old packetPage with the new packetPage.
4. The SPU software requests the PMU to de-allocate the space associated to the old packetPage.
- 5.

It can be readily seen by one with skill in the art that many packet move operations would certainly require considerable processing resources.

To enable allocation of extra memory space to insure space for a packet to grow during processing, two pre-boot configuration registers (not shown) are provided. These are termed a TailGrowthOffset register and a HeadGrowthOffset register. These configuration registers in a preferred embodiment have a digital value indicating the number of 16 byte-wide lanes assigned to head and tail offset. For example, in one embodiment the HeadGrowthOffset is 4 bits, to allow for 16 16-byte lines, and the TailGrowthOffset is 6 bits to account for 63 16-byte lines. The goal of the present invention is to provide suitable memory space for a packet to grow

during processing such that consecutively stored data packets do not overwrite each other, or that packet move operations can be sharply reduced or avoided altogether.

Figure 51 is a block diagram illustrating a portion of memory space 5100 for holding consecutively stored data packets according to an embodiment of the present invention. Packet memory 5100 is analogous to packet memory 5000 described with regards to Fig. 50 above with the exception of how data packets are stored.

In this example data packet B (5001) has a portion of memory allocated thereto in addition to the portion allocated for the actual size of packet B. A portion represented herein by the letter h is allocated for possible head growth of packet B during processing. A portion represented herein by the letter t is allocated for possible tail growth of packet B. The actual data size of packet B can be represented by the letter S (not shown). Therefore, the amount of memory 5100 allocated for data packet B can be expressed as  $M = (s + h + t)$  where  $M$  is the total memory allocated. The same description above applies equally to packets A and B.

Data packets A and C exhibit the same buffer zones h and t bounding s (original packet size). It is noted herein that in actual practice, the values of h and t are pre-configured at boot time and remain constant values for all incoming data packets whether stored in hardware or software-controlled memory. For example, the value of h is the same for packets A, B and C. Similarly, the value of t remains constant for all packets as well. The only variable is s, which is the starting size of any packet before processing. However, in another embodiment higher levels of granularity may be introduced. For example, all VoIP packets may have specific h and t values while all other packet types have constant h and t values different from the VoIP values. In this case, additional configuration registers would be

provided for isolating specific types of data packets. This embodiment, of course, assumes that the packet type is known at the time of memory allocation for storage.

It is described above with reference to S/N 09/737,375, that the PMMU is responsible for memory allocation and for determining if and where in local packet memory a packet will be stored. The PMMU handles memory allocation in hardware-controlled LPM. Software in the SPU handles memory allocation in the event that it is determined that a packet does not fit in LPM. Software can access the offset registers containing h and t values, so that the allocation of extra memory space for packet growth is practiced for all of memory 5100.

In actual practice, the values of h and t are configurable as follows for application to all preprocessed data packets. A configuration register HeadGrowthOffset (h) has a minimum value of 0 and a maximum value of 240 bytes of memory in a preferred embodiment. Another configuration register TailGrowthOffset (t) has a minimum value of 0 and a maximum value of 1008 bytes of memory in the same embodiment. These values are stored in the previously mentioned registers associated with h and t during boot. When a data packet arrives, the PMMU consults the registers to read out the values of h and t and compute the values along with the value of s (original packet size) to produce  $M$ . The PMMU will look for a space in memory 5100 that can accommodate the total computed size  $M$ . It is noted herein that the PMMU still only needs to generate a single offset to store the data packet because h, t, and s are all known at the time of page allocation. The generated offset is from the initial address where the packet is to be stored.

Packets in many cases will not grow to fill the allocated memory space h and t. In some cases, a packet may actually shrink after processing.

Moreover, consecutively stored packets may exhibit differing growth amounts. The only time a packet move operation is required is if the head grows more than h or the tail grows more than t. If SPU software is charged with storing a data packet, the PMU can provide the required values of h and t to the SPU because it has already consulted the registers when determining that a particular packet did not fit memory 5100. In a case where SPU software is solely responsible for storage of a packet, then it may access and read the registers independently from PMU hardware.

In one aspect of the present invention, there may be a plurality of different registers holding h and t values for a plurality of differing types of data packets. Also in one embodiment, changed values of h and t may be introduced into the registers while the system is running and processing data.

In this case new values may be determined dynamically, based on such as historical performance. Keeping historical performance records can be accomplished in order to refine predicted packet growth for differing types of data packets.

It has been previously described that at some point processing is complete, and the packet may be downloaded from the LPM to be sent out of the system. This downloading is a function of the PMMU. Clearly, when download commences, the PMMU has to read the header growth offset for a packet, to determine where the packet really starts in LPM. For example, if a packet, ready for download, was stored with a header growth offset, but the packet header has not grown or shrunk in processing, then the first data for the packet will be at the beginning of the first page allocated, plus the header growth offset.

If the header for a packet has grown or shrunk, the software needs to notify the PMMU of the magnitude of the header change so the PMMU can find the beginning of the packet in LPM. For example, if the header grew by

X, the first packet data will be at the beginning of the first page allocated, plus the offset, less the amount of the growth, and so on.

Additionally, in those embodiments wherein offsets may change dynamically, that is, on-the-fly, there are mechanisms to ensure that offset changes are made only at times that the changes will not create confusion regarding the actual starting point of a packet in memory at download time.

Accordingly the claims that follow should be accorded the broadest interpretation.

5

10  
15  
20  
25  
30  
35  
40  
45  
50  
55  
60  
65  
70  
75  
80  
85  
90  
95  
100